The Pragmatic Programmers

TURING 图灵程序设计丛书

Debug It!

Find, Repair, and Prevent Bugs in Your Code

软件调试修炼之道



[美] Paul Butcher 著

曹玉琳 译

- 亚马逊全五星畅销图书
- 资深专家经验总结
- 问题重现→问题诊断→缺陷修复→反思



人民邮电出版社
POSTS & TELECOM PRESS

Debug It!
Find, Repair, and Prevent Bugs in Your Code
软件调试修炼之道

“Paul出色阐释了调试各个阶段所涉及的技术、思维、心理等方面的问题：首先要预防缺陷发生，然后诊断和修复缺陷，并确保相同的缺陷不再发生。应用本书的任何一种思想都有助提高软件项目的整体质量。当然，Paul从心理学的角度很好地阐述了技术问题，这也是本书的一大特色。”

——Frederic Daoud，*Stripes: ...and Java Web Development Is Fun Again*作者

“我由衷地向大多数软件工程师推荐这本书，特别想把它推荐给力图实现最佳实践的团队领导。”

——Allan McLeod，Isaacc软件公司创始人兼首席技术官

“本书在设置调试场景、获得正确思维模式方面讲了很多，同时也谈到当缺陷发生及解决后可能出现的连带问题。单是其中的轶事就非常值得一读，你会看到软件工程师们在试图了解真正奇怪的缺陷时是多么百折不挠。”

——Jon Dickinson，*Grails 1.1 Web Application Development*作者

“长时间以来，调试技术一直是一项‘民间艺术’。现在有人把所有这些经过实践验证的技术归纳总结起来，这真是一件非常了不起的事。当你对如何创建优秀软件的过程一无所知时，这本书可以很好地给你指明方向。当你有了断言、日志、重构等好方法时，你会觉得自己就是福尔摩斯，没有什么疑案是破解不了的。”

——Craig Riecke，《精通Dojo》作者

“这本书就像《程序员修炼之道》的姊妹篇，它将注意力集中在调试的技巧上。”

——Ian Dees，*Scripted GUI Testing with Ruby*作者

“Paul Butcher讲述了我们早该关注的调试方法。对于每一位软件工程师来说，调试是一项基本活动，而大多数人仍然在靠直觉和猜测来进行调试。Paul朴实无华的写作风格掩盖不住他在技术方面的超群素养。如果不了解调试，那你只能是一名软件工程师，而永远无法成为黑客级程序员。”

——Bill Karwin，Karwin软件公司软件工程师

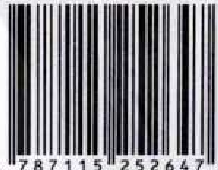
The Pragmatic Programmers

图灵网站：www.turingbook.com 热线：(010)51095186转604
反馈/投稿/推荐信箱：contact@turingbook.com
有奖勘误：debug@turingbook.com

分类建议 计算机/软件工程

人民邮电出版社网址：www.ptpress.com.cn





ISBN 978-7-115-25264-7
定价:32.00元

版 权 声 明



前　言

我一直想不明白，为什么关于调试的书这么少。其他软件工程方面的书可谓汗牛充栋，设计、代码结构、需求获取、方法学，林林总总，然而有关调试方面的书却既没有什么人写，也没有什么人出，我希望这本书有助于改善这种情况。

如果写代码，那么在某些时候肯定要调试代码（也许很快就需要调试）。调试比其他任何过程都更需要动脑，它不发生在调试器或代码中，而是形成于大脑之中。找到并理解问题的根源，才能进行其他的工作。

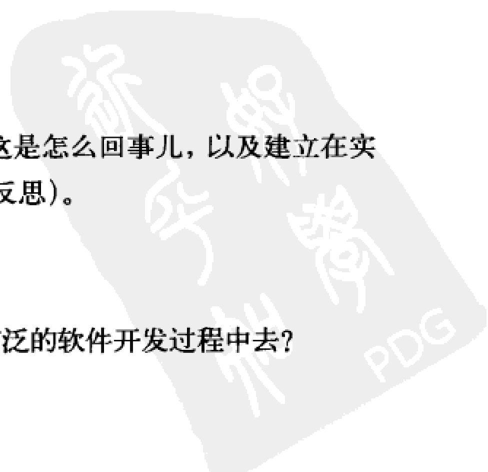
多年来，我有幸在多个软件领域与许多出色的团队共事。我曾做过位片处理机微编码的所有抽象层次上的工作，涉猎过设备驱动程序、嵌入式代码、主流台式机软件和网络应用程序。我希望能够通过这本书总结一些我从同事那里得到的经验教训。

关于本书

本书分为三部分，每一部分都详细阐述了调试的某一方面。

第一部分　问题的核心

这部分介绍了实证方法，即借助软件特有的功能向我们展示这是怎么回事儿，以及建立在实证方法之上的核心调试方法（问题重现、问题诊断、缺陷修复、反思）。



第二部分　从大局看调试

我们怎样发现存在需要修复的问题？又怎样将调试融入到更广泛的软件开发过程中去？

第三部分　深入调试技术

这部分将关注一些高级的话题。

- 尽管本书前面讨论的方法适用于所有缺陷，但某些类型的缺陷还是需要特别对待。
- 调试应该早早启动，不要等到受困的客户愤怒地打电话来才开始调试。我们能够提前采取什么措施和流程来解决客户可能提出的问题呢？
- 最后讨论如何避免一些常见的缺陷。

致谢

直到开始写这本属于自己的书时，我才认识到致谢这一节多么重要。我的名字得以出现在封面上，但是没有很多人给我的帮助和对我的宽容，这本书是不可能完成的。

感谢参加本书电子邮件讨论列表并提供灵感、批评和鼓励的每一个人：Andrew Eacott、Daniel Winterstein、Freeland Abbott、Gary Swanson、Jorge Hernandez、Manuel Castro、Mike Smith、Paul McKibbin和Sam Halliday。要特别感谢Dave Strauss、Dominic Binks、Marcus Grber、Sean Ellis、Vandy Massey、Matthew Jacobs、Bill Karwin和Jeremy Sydik，他们允许我将他们的轶事和见解与你分享。还要感谢Allan McLeod、Ben Coppin、Miguel Oliveira、Neil Eccles、Nick Heudecker、Ron Green、Craig Riecke、Fred Daoud、Ian Dees、Evan Dickinson、Lyle Johnson、Bill Karwin和Jeremy Sydik，感谢他们花费时间为本书做技术审查。

感谢我的编辑Jackie Carter，他耐心地指导我，让我这个第一次写书的人摸到了写书的门道。感谢Dave和Andy给了我这次机会。

我要向工作在Texperts的同事们致歉，他们不得不忍受我喋喋不休一味地谈论这本书的内容（不要担心——我很快就会有一辆新的赛车，那时你们又必须忍受我喋喋不休地谈论它了）。还要向我的家人致歉，为我在漫长的夜晚和周末不能和家人在一起而道歉，感谢他们对我的支持。

最后，感谢所有我能够荣幸地与之共事的人。软件开发生涯中最好的一面就是软件人的器量，在一个好的团队中工作是一件特别的幸事。

Paul Butcher
2009年8月
paul@paulbutcher.com

目　录

第一部分　问题的核心

Part 1 第一部分

问题的核心

本部分内容

第 1 章 山重水复疑无路

如果你的软件不能正常工作，该怎么做呢？

一些开发人员似乎有窍门能够准确无误地找出发生缺陷的根本原因，而另一些开发人员似乎总在漫无目的地寻找原因却得不到确切的结果。是什么使他们之间存在着这样的差异呢？

在这一章中，我们将仔细探究一种调试方法，这种方法在专业的软件开发中已经被反复证实。它绝非灵丹妙药，它仍然依赖于调试者的智慧、直觉、探查缺陷的技巧，甚至一点儿运气。但是，它会使你的努力更加有效，避免做无用功，并且能够尽快地找到问题的核心。

具体来说，我们将介绍以下内容：

- 调试与排除缺陷的区别；
- 实证方法——借助软件本身来告诉你现在的运行状态；
- 核心调试过程（问题重现，问题诊断，缺陷修复，反思）；
- 做最先应该做的事——在深入调试之前应该先考虑的事情。

1.1 调试不仅是排除缺陷

试问一个没有经验的程序员什么是调试，他可能回答调试就是"找到一种修复缺陷的方法"。事实上，这仅仅是调试诸多目标中的一个，甚至不是最重要的目标。

有效的调试需要采取以下步骤。

(1) 弄清楚软件为什么会运行失常。
(2) 修复这一问题。
(3) 避免破坏其他部分。
(4) 保持或者提高代码的总体质量（可读性、架构、测试覆盖率、性能等）。
(5) 确保同样的问题不会在其他地方发生，也不会再次发生。

其中，目前最重要的是首先查明问题的根本原因，这是一切事情的基础。

理解万岁

一些没有经验的开发人员（遗憾的是，有时就是我们中本应知道得更多的那些人）经常完全忽略问题诊断这一过程，而是往往立即采取他们认为能够修复程序的措施。如果他们走运，只是修改了的程序运行不了，他们所做的一切是在浪费时间而已。真正的危险是修改了的程序可以运行，或者看来可以运行，因为这时开发人员在一知半解的情况下改变了源程序。他或许修复了缺陷，但是实际上很可能掩盖了潜在的根本原因。更糟糕的是，这种改变可能会导致新问题——破坏一些曾经可以正确运行的程序。

浪费时间和精力

几年前，我在一个拥有很多具有丰富经验的天才开发人员的团队中工作。他们的经验大部分来自 UNIX 操作系统，但是当我加入这个团队时，他们正处于将软件移植到 Windows 操作系统的后期阶段。

在移植的过程中，他们发现了一个缺陷，就是同时运行多个线程时出现了性能问题。某些线程突然死掉，而另外的线程在正常运行。

所有的程序在 UNIX 操作系统下运行正常，问题很显然是 Windows 操作系统破坏了线程，因此他们决定定制一个线程调度系统，以避免使用操作系统所提供的调度系统。显然这是一项很繁重的工作，但是我们的团队有能力完成这项工作。

我加入这个团队时，他们正在以各种方式实现这项工作，果然，线程不再突然死掉。但是，线程调度是难以捉摸的，即使变化引起了很多问题（比如使整个系统运行变慢），这些线程仍然能够继续工作。

我对这个缺陷很好奇，因为在以前的 Windows 线程编程中，我从来没碰到过这样的问题。略作调查后我们发现，性能问题事实上是由于 Windows 实现了**动态线程优先**。这个缺陷其实通过禁用一行代码就可以修复（即调用 `SetThreadPriorityBoost()` 函数）。

这件事情告诉了我们什么？该团队没有深入调查所看到的现象，就下结论，认为 Windows 线程被破坏。在某种程度上，这可能是文化问题，Windows 在 UNIX 黑客中声誉不佳。不过，如果他们花点儿时间找出问题的根源，就能省去大量的工作，避免引入使系统效率低下并易产生错误的复杂因素。

如果不首先弄清楚缺陷的真正根源，我们就没有遵循软件工程的原则，而是一头扎进巫毒编程（voodoo programming）[1]或者巧合编程（programming by coincidence）[2]里了。

1.2 实证方法

可以采用很多不同方法去实现你所探查的目标。只要你选择的方法让你更接近目标，它就达到了目的。

话虽如此，在大多数情况下，一种特别的方法——实证方法，是迄今为止最有效的方法。

> **构建实验，观察结果。**
> *Construct experiments, and observe the results.*

实证依赖的是观察和经验，而不是理论和纯逻辑推理。就调试而言，这意味着直接观察软件的行为。是的，你可以阅读全部的源代码，并用纯理论去了解软件的运行状况（有时你可能没有其他选择），但是这样做通常没什么效率，而且非常危险。通过仔细地构建实验环境并观察软件的运行状况，你可以更有效地找到问题。这样做不仅仅更快捷，而且通过观察可以使你重新审视自己关于软件运行还有哪些错误的假设。软件本身就是你的工具箱中最强有力的工具——就让它来告诉你运行状况是什么样的吧。

软件的本质

软件是非常了不起的产品。有时，或许是因为我们一直与它一起工作，所以才会忘记它是多么地了不起。

人类的经验中有一小部分具有可塑性，允许我们毫无限制地自由发挥聪明才智和创造力。当然，软件是确定性的（有少数例外，后面会提到）——下一个状态完全取决于当前的状态；更为关键的是，只要需要，我们就可以进入各个状态。

相对于传统的工程学，我们太幸运了。你认为一名F1工程师会瞬间停止每分钟19 000转的引擎来检查它的每一个细节吗？例如，他能查看每个组件在压力下的精确状态，动态记录点火时燃烧室内前方火焰的位置和形状吗？

而我们却能够凭借这种技巧检查我们的软件，这就是为什么在调试时实证方法特别强大的原因。

---

① “在晦涩难懂、令人发指的系统、程序或算法中，人们不能真正地理解，使用的是一种猜测或手册方式。其含义是，该技术可能无法工作，如果它没工作，人们永远不知道是为什么。”摘自 *The Jargon File*[ray]。

② 见 *The Pragmatic Programmer*[HTOO]。

下一节中描述的方法将会利用实证法去提供一个结构化的手段来集中找出缺陷。

1.3 核心调试过程

调试过程的核心包括以下四步。

问题重现 找一个可靠并简洁的方式来按需求重现问题。

问题诊断 提出假设，并通过实验来测试它们，直到找出引起缺陷的潜在原因。

缺陷修复 设计和进行一些修改来修复问题，不要引入回归问题，保持和提高软件的整体质量。

反思 吸取教训。哪里出了问题？是否还有其他类似的问题需要修复？怎样做才能确保同样的问题不再发生？

> 调试是一个反复的过程。
> *Debugging is an iterative process.*

如图 1-1 所示。一般来说，这些步骤是按顺序执行的，但这并不是严格的“瀑布”模型。虽说你肯定不希望直到重现了问题或者设计了修复计划后才启动诊断，而后才明白所发生的问题，但这是一个迭代的过程。在问题诊断中学到的知识可能会告诉你如何提高重现问题的水平，或者在软件修复中学到的知识可能会使你重新考虑你的诊断结果。

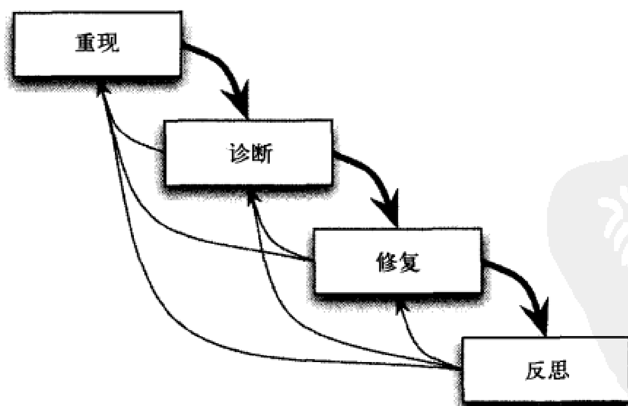


图 1-1 核心调试方法

在下面的章节中，我们将详细研究这些步骤。而在此之前先来了解一些基本原则。

1.4 先澄清几个问题

径直深入讨论可能更诱人，然而还是值得花点儿时间来说说一些基本原则，以确保之后的讲解能够顺利进行。

1.4.1 你知道要找的是什么吗

> **发生了什么？应该发生什么？**
>
> *What is happening, and what should?*

在开始重现问题或者推测问题产生的原因之前，你需要明确地了解到底发生了什么。同样重要的是，你还要知道正常情况下应该发生什么。假如你是根据一份正式的缺陷报告来工作，那么这个报告应该包含了所有你需要的信息。（我们将在第6章详细讲解缺陷报告。）花点儿时间仔细阅读一下，确保你充分理解它。

假如你没有正式的缺陷报告（可能你正在解决一个备感困惑的缺陷或者这个缺陷是在不经意间提交的），那么更应该先暂停你的工作，要确保在继续工作之前真正地了解整个程序。

请记住，缺陷报告本身的错误不会比其他文档更少。仅仅是因为缺陷报告说应该这样运行而不应该那样运行，就能真正地符合软件的规格说明书吗？如果它没有明确地说明应该怎样运行，那么在你彻底弄清楚之前不要做任何改变——否则有可能会把正确的改为不正确的，仅仅听信缺陷报告是不会有帮助的。

不要轻信缺陷报告

曾经有一次，我修改一个非常简单的缺陷——生成的报告说程序没有考虑夏令时间，因此当时钟变化时出现了错误。我快速地修复了此缺陷，然后转向解决下一个问题。

然而稍后出现了另一个缺陷，缺陷是账目不能收支平衡。报告生成的数量与我们从供应商那里得到的发票的数额不一致。

果然，事实证明，这些票据没有考虑夏令时间，从而导致了差异的产生。再查以前的记录发现，我们一年前就已经遇见了这个问题，那时我们故意忽略了夏令时间。[1]

显然，这个问题不是软件没有听我们指挥，而是我们自己不知道想让软件做什么。因为报告是在不同的环境下使用的，在某些情况下，夏令时间是需要考虑的，而另外一些情况下，夏令时间是不需要考虑的。正确的解决办法是在报告中增加选项，允许用户选择。

---

① 顺便说一句，最开始对源程序做出改变的开发人员如果能够加上简单的代码注释，说明为什么在这种情况下忽略夏令时间，让我们明白是有意忽略夏令时间的，就会减少很多麻烦。

1.4.2　一次一个问题

有时，面对几个问题时往往会并行处理，这似乎挺带劲。如果缺陷都发生在同一区域，这种做法尤其常见。但是请不要这样做。

调试一个缺陷已经很困难了，使情况更为复杂是没有必要的。无论怎么小心，你为了查找一个缺陷所进行的实验很有可能会以某种方式诱发另外的缺陷。这样很难弄清楚究竟发生了什么。此外（正如我们4.5节所讲），你最终要签入修复程序时，会希望每做一次逻辑修改就签入一次，而如果你同时修改多个缺陷，那么这就很难实现。

有时，你会发现你认为一个缺陷是由某个原因引起的，而事实上它却是由多个原因引起的。通常情况下，当你处于思维迟钝的状态时，这点更明显——奇怪的是这种现象似乎没有明确的解释。要进一步探讨，你可以参阅3.4节。

1.4.3　先检查简单的事情

很多缺陷是由于简单的疏漏而引起的。因此，有时你会面对非常微妙的事情，但请千万不要忽略简单的事。

由于某些原因，开发者似乎有一种感觉——不得不亲自做一切事情。在NIH（Not-Invented-Here，非我发明症）中表现得最为明显，这种感觉使得我们想要自己实现程序，但其实在其他地方已经存在一个很好的解决方法。这种错误的观念在调试方面的表现就是你必须亲自调试你遇到的每一个问题。

问问团队中的其他成员，是否以前也遇到过类似的问题，这样做可以大大降低成本，而且很有可能避免浪费大量的精力。如果你工作在一个并不熟悉的领域，这样做尤其重要。

Subversion之困惑

肖恩·埃利斯

这周，我的一个新伙伴正在受SVN export问题的困扰。这是一个致命的打击——服务器及其工作站拥有相同版本的SVN，但是状态却不相同，存在很多潜在的问题。

终于，他崩溃了。他问我，用这个特有的命令是否有问题，并剪切粘贴了SVN中的命令行给我。

“是的，有问题。”我说。Apache库中有这样的缺陷，在路径中用../../..标记是错误的。两秒钟后，我们确定这就是真正的问题所在。几分钟后我们确定了服务器端有

不同版本的 Apache 运行时 DLL。

当然，几个月前很多潜在的问题就相继发生，而这是第一次发现这个缺陷。

因此，沟通总是重要的——不沟通容易陷入奇怪、微妙、令人生厌且难以描述的方式中。古语说得好："站起来，问问是否有人以前见到过此类问题。"

下一章，我们将详细讲解这个过程的第一步——重现问题。

1.5 付诸行动

- 一定要做到以下几点。
  - 找到软件运行异常的原因。
  - 修复问题。
  - 避免破坏其他程序。
  - 保持或提高软件整体质量。
  - 确保不在其他地方发生同样的问题，确保这种问题不重复发生。
- 利用软件自身来告诉你发生了什么。
- 每次只解决一个问题。
- 确保你知道自己要找的是什么。
  - 正在发生什么？
  - 应该发生什么？
- 先检查简单的事情。



第 2 章

重现问题

正如上一章所述，运用实证方法进行调试可以充分利用软件的独特能力，来告诉你软件运行的状态，而发挥这种能力的关键是找到能够重现问题的方法。

在这一章中，我们将介绍以下内容：

- 为什么重现问题是如此重要；
- 如何对你的软件施加必要的控制，找到一个重现问题的方法；
- 怎样才能做好问题重现，以及如何通过反复优化来达到这个理想目标。

2.1 重现第一，提问第二

为什么重现问题如此重要？因为如果你不能重现问题，那就几乎不可能取得进展。原因如下。

- 实证过程依赖于我们观察存在缺陷的软件执行的能力。我们应该首先设法使软件表现出它的缺陷，如果不能，那么这就如同我们丢失了军火库中最强大的武器。
- 即使你设法提出一个为什么软件可能出现缺陷的理论，但是如果你不能重现该缺陷，那么又如何证明你的理论呢？
- 如果你认为你已经实施了软件修复，那你又如何证明确实解决了问题呢？

不仅重现问题很关键，而且更为关键的还在于这是你应该做的第一件事情。如果未能成功地重现该问题就开始修改源代码，那么你所做的修改可能会掩盖一些问题或者带来其他的问题。[①]

那么，究竟如何开始这个调试的关键阶段呢？

---

① 这类似于测试优先开发中的原则，没有一个失败的测试，你就不应该写任何新的代码。在这种里，你的“失败的测试”就是被重现的缺陷。

2.1.1　明确开始要做的事

要做的第一件事很简单，就是按照缺陷报告描述（或提示）的步骤来做。

这对即使是只有一行的缺陷报告也同样适用。我曾经看到开发人员拒绝一份类似于“取消更改密码对话框出错”的缺陷报告，声称“需要更多的信息”，甚至不去尝试重现这个缺陷。虽然我们都曾经为缺少关键信息的缺陷报告感到郁闷，但有些错误跟你运行的操作系统、软件的当前配置、当时运行软件时做了些什么，等等诸如此类应在缺陷报告模板中包括的信息其实关系不大。有时候打开更改密码对话框，然后点击取消，该软件就有可能会崩溃，这样你就可以马上开始诊断，而不必打扰用户来获得更多的信息。就算查不出问题，也不会浪费你很多时间。

如果这个简单的方法没有效果，那么缺陷本身会提供一些有用的线索来告诉你下一步该怎么做。

2.1.2　抓住重点

要做好问题重现就要抓好控制。如果你控制了所有相关的因素，就能重现你的问题。当然，方法就是确定哪些因素是和当前缺陷有关的，你要如何去设置它们，并找到一种方法去实现。

作为一名开发人员，你的环境和用户的环境是不同的。你使用的是最新的源代码，而用户运行的可能是在几周、几个月甚至是几年前编译的代码。你的配置与用户的也会有所不同，你的网络环境、使用的外设等都可能不一样，或多或少的这些差异会妨碍缺陷的重现。因此，你的第一个任务，就是找到并消除这些差异。

很多因素可能会悄然地影响软件的运行。在大多数情况下，它们之间是很少相关的。那么怎么知道首先应该集中精力解决哪个因素呢？

你需要控制的因素可以归结为以下三个方面[①]。

软件本身　如果缺陷存在于最近修改的地方，那么你应该首先保证你调试运行的软件和缺陷被提交时使用的软件是同一个版本，因为这是保证问题重现的起码要求。

软件的运行环境　如果要与外部系统（某些特定的硬件，或者一个远程服务器）进行交互，那么可能需要确保你使用的是相同的外部系统。

你提供的输入　如果一段代码的运行情况受软件的配置影响很大，而缺陷又与这段代码有

① 这些方面之间的界限也不是很明确——一个人的开发环境可能是另一个人的输入环境。对此不必太在意。如何对你需要控制的因素进行分类是无关紧要的，只要你成功地控制就可以了。

关，那么你就应该使用用户的配置来进行调试。

在下面的几节中，我们将详细讲述每一个方面。

2.2 控制软件

如果你使用最新的源代码而不是用户正在运行的版本，无法迅速地重现缺陷，这当然有可能是因为缺陷已经被修复了。但是，你不能做出这种假设，因为很有可能缺陷仍然存在，只不过是以另一种不同的形式存在。只有当你已经完成了诊断之后，才能确定是不是没有缺陷了，而诊断的第一步就是重现问题。

仅仅通过编译同样的源代码并不能保证你会运行同样的目标代码。你还需要确保你使用相同的编译器，以相同的方式进行配置，使用相同的运行时环境、相同的库文件，以及相同的集成到你软件中的任何第三方代码。

当然，即使是使用相同的工具，如果你不准确地按照执行顺序并以最初构建软件时使用的配置来使用它们，也不会得到正确的结果。确保你做得正确的最好办法就是创建一个自动化的构建过程，我们将会在 9.3 节中详细讨论。

2.3 控制环境

软件环境的构成取决于你的软件类型。对于传统的桌面软件，操作系统可能是最相关的因素；对于 Web 软件，浏览器可能是最相关的因素；对于网络软件，与它进行通信的软件可能是最相关的因素；对于嵌入式代码，它所使用的硬件可能是最相关的因素。

尽管存在这些差异，关键还是首先要知道缺陷出现时软件所使用的环境，我们将在 6.1 节中讨论如何实现这个目标。而且，你也需要方便地访问所有可能的环境，以便在任何一个相关的环境中进行测试。

一些人有幸在一个与运行环境相同（或足够类似）的开发环境中工作，这意味着我们也许可以很容易地在开发机器上重现问题。（反之，如果一切都在我们开发的机器上运行，那么我们可以确信软件部署后会很好地工作。）但是，如果你针对多个平台编写嵌入式软件，或者在笔记本上开发而在服务器上运行，那么你就必须想办法来构建一个相同的运行环境。

重现不同的运行环境对后勤部门来说曾是一个噩梦。从地面到天花板，整个房间塞满了不同品牌型号的计算机，使得我们可以使用每个版本的硬件和操作系统，这样的软件开发场所一度也很常

见。有两件事情在很大程度上帮助我们解决了这个问题。第一个就是硬件抽象——计算机图形卡可能大大影响软件运行的日子已经一去不回了，谢天谢地。[①]第二个就是虚拟机——几乎不用费力就可以在一台计算机上同时运行很多的操作系统和进行不同的配置。如果你正在使用的是跨平台的软件，那么虚拟机的作用会非常明显，它在其他应用环境下也是很有用的。

例如，你正在编写 Web 软件，可能需要支持各种浏览器，而且可能每个浏览器都有几个不同的版本。实现这一目标的最简单方法（特别是在单一操作系统上安装多个浏览器的多个版本是很有难度的）可能是使用若干个不同的虚拟机，每个虚拟机配置一个不同的操作系统和浏览器。

另一个例子是，你正在编写一个要同时运行在多台计算机上的软件——你的软件可能需要被部署到一组机器中？如果是这样，你可以在一台开发用的机器上并行地运行几个虚拟机，从而创建一个“虚拟数据中心”。

最后，请记住，软件环境包括了可能影响软件运行的所有因素。有时，如下面的故事所述的，可能会包括一些我们认为不可能的因素。

> **软件环境包括可能影响软件运行的所有因素。**
>
> *Your software's environment is anything that might affect its behavior.*

都是精灵惹的祸

Dave 的工作是为打印机设备编写驱动程序。经过几个星期的工作，他觉得程序已经编写完成，就把它交给楼上的测试人员进行测试。很快，测试人员发现了一个间歇性缺陷——在输出中出现虚的水平线。

想尽一切办法，Dave 还是没能重现该问题。他一页一页打印，没有找到一处这样的错误。我们开始寻找开发环境和测试环境的差异，但也没有任何效果。我们甚至把整个测试系统从楼上搬到楼下。我们选择了一个能够重现问题的系统，几个人沿一级级台阶把它搬到楼下——可是它的运行没有任何问题。

Dave 认为缺陷是由一群住在楼上的精灵造成的，他们干扰了打印机的内部工作。令人诧异的是，结果他的想法被证明是最接近事实的。

我们的办公室位于剑桥郡乡村，是一个非常好但又古老的房子。这是一个工作的好地方，但有它的缺点。其中一个缺点就是它的线路问题，这些线路虽然不像房子那么古老，但是比大多数工作人员的年纪都大。原来，楼上电力系统配置得不是很好，电压的随机波动足以导致我们得到不同的运行结果。

① 除了少数专业领域以外，比如游戏。

2.4 控制输入

软件的输入数据可能是磁盘上的文件、一系列的用户界面操作，或者是来自第三方服务器或设备的响应。无论何种形式的输入，关键是要首先找出输入数据，以便你准确地重放它们。

如果你运气好，相关的输入将被明确地记录在缺陷报告中，但情况并非总是如此。对你来说，一个缺陷报告需要列出运行过程中的每一个步骤，但你的客户不太可能认识到这样做的重要性。或者他们可能会在描述中加入他们自己的先入之见（而这种意见可能与实际运行的状况相去甚远）。

即使用户认认真真地汇报他们所做的一切，仍然可能是不够的。通常对于最终用户而言，很多重要的细节不那么显而易见，甚至根本无法获得。例如，该缺陷可能是由细微的时间差异导致的，或者是由后台接收第三方系统的输入而产生的。

如果你无法获得需要的所有信息，就有两种选择。或者推测一下可能的输入是什么，或者把它们记录下来。

2.4.1 推测可能的输入

推测正确的输入来重现问题的出发点是假设问题确实存在，然后运用逆向工程的方法推测出能够导致问题的必要条件。

1. 回溯工作

通常我们知道发生了什么事，但并不很清楚为什么会发生。

例如，我们有一个缺陷报告，指出了程序因为一个空引用而崩溃。我们可以从错误信息得知哪一行源代码发生了空引用，但我们不知道程序以什么顺序执行到这一行。

我们所能做的就是回溯我们的工作。我们可以推断，如果变量 a 在那行代码中是空的，那么就必然意味着一个不存在的标识符被传递给方法 b()，而这又意味着必须使用特定的输入值来调用 c……

如果你运气好，这个逻辑可以直接重现问题。即使不能完全重现，它也可以提供有用的线索，再加上其他的异常现象，就可以排除某些可能性。

2. 探测可能的输入值

即使缺陷报告中的一系列的输入不能重现问题，但与它们相类似的输入极有可能会重现问

题。可能缺少某一个重要的步骤，或者他们说点击了那个按钮，但实际上却是点击了这个按钮。在这种情况下，你可以通过查看报告中的相似步骤来找到正确的执行顺序。

在这里，你熟悉的很多软件测试技术将会派上用场，特别是边界值分析法和分支覆盖法。

边界值分析 经验表明，输入值取值范围的边界是最有可能出现错误的地方。如果你的软件应该在输入值为10以下时做一件事情，而为11或以上做另一件事情，那么当你输入10或11时就极有可能会出现错误。其他常见的边界条件是长度为零的输入值，或者从正数变为负数的输入值。

分支覆盖 分支覆盖相当于白盒测试中的边界值分析（边界值分析是黑盒技术）。[①]如果你不能通过一系列的输入来重现问题，那么可以尝试创建一些输入，使用这些输入可以覆盖同一代码块中的不同分支。

有效地识别能重现问题的输入顺序需要你转变思路——你不必证明系统工作正常，而要证明它不正常。

还有其他的方向

在《程序员修炼之道》[HT00]中，Andy讲述了一个同事的故事，他在竭尽全力重现图形应用程序中出现的问题。缺陷报告说，每当使用一种特定的画笔在屏幕上划一笔时，软件就崩溃了，但是他坚定地认为一切运行良好。

几天后，大家慢慢冷静下来。他们最终发现，每当他“测试”画笔时，他总是从左下的位置划到右上的位置（换句话说，同时增加x和y的坐标）。而当他向另一个方向划时，立刻就出现了问题。

3. 利用错误条件

编写代码时，人们往往会将精力集中在进展顺利的方面，这是人的天性。我们心里有一个特定的目标，而且往往集中精力去实现这个目标，而不操心那些可能导致错误发生的方法。加之测试错误条件可能很麻烦，结果就是错误条件可能诱发一系列缺陷。

当试图重现一个问题的时候，考虑一下是否有一些错误条件出现在运行的过程中，并去解释为什么问题会发生。然后，想想如何使错误条件表现出来或者模拟出来，并且是否可以使你重现问题。

---

① 黑盒技术，不必了解测试系统的内部结构就可以得到测试用例。与此相反，白盒技术会利用对系统内部结构的了解来创建测试用例。

4. 引入随机性

选取一系列不同输入值的一种方法就是引入一些随机输入值。比如，如果你寻找一个似乎依赖于时序细节的缺陷，那么把随机输入值引入其中会大大增加缺陷出现的机会。

模糊测试包括为程序提供随机数据（或称模糊），一个模糊测试器可以使调试过程自动进行（见 A.4 节）。模糊测试器要么通过生成模糊器要么通过变异模糊器生成模糊数据。

生成模糊器　生成模糊器利用一个数据模型生成输入值，它使用新的数据或者按某些方式将已有数据组合起来。这种数据模型融合了对被测试软件的理解，可以提高发现问题的几率。

变异模糊器　变异模糊器是由一个熟知的、可以根据一系列规则进行修改的模板生成的。这些规则的出发点是要增加根据结果输入来揭示缺陷的能力。

所有模糊测试器的一个关键特征是，它们可以重新创建任何曾经生成的输入，以便问题出现时，能够随意重现问题。

在推测重现问题时需要使用的数据的过程中，请记住你需要验证与缺陷报告不符的结论。你找到了一种使软件出现缺陷的方法，并不意味着你找出了缺陷报告中提到的缺陷（尽管你已经清楚地发现了一个需要修正的缺陷）。

2.4.2 记录输入值

推测正确的输入以重现问题的另一种选择，是通过日志来直接记录输入值。如果你的软件已经内置了日志功能，那么就可以很简单地让用户打开日志功能并把结果发送给你。或者，你可以给他们发送一个自定义的软件或者其他一些日志解决方案（例如一个调试程序或代理程序）。不管你决定采用哪种解决方案，都应准确了解用户真实的操作，这是最有价值的。

1. 日志

获得日志的最简单方法就是，在整个代码中正确地放置对 `System.out.println()`或类似方法的调用。事实上，这种简单的方法可能会满足你所有的需求。但是如果你的日志需求确实十分复杂，你就应该考虑从诸多可用的日志框架中选择一个使用（见 A.3 节）。

日志框架可以免费为你提供大量有用的功能。

- 能够在需要的特定地方打开或关闭日志功能。
- 不同的日志级别，能使你对生成的日志数量进行微调。在正常运行期间，或许你只记录了软件发生致命错误时的日志，或者仅仅大概记录了软件能够做什么而没有任何的细节。

但是当你需要的时候，你可以提高日志的级别来生成更多的信息，甚至可以精确地记录哪些函数被调用了，是在什么时候被调用的，传递了哪些参数。

- 日志信息可以包含很多非常有用的信息，比如这些信息与哪个日志级别或模块有关，甚至可以精确地获得源文件的行数。
- 可以帮助分析日志文件的标准工具。
- 自动记录某些事件，比如未处理的异常。

实际中是如何使用日志框架的呢?下面是一个使用 java.util.logging 框架的 Java 类的例子。

```
import java.util.logging.Logger;

public class Dispatcher {
❶  private static final Logger log = Logger.getLogger(Dispatcher.class.getName());

  public static void dispatchLoop() {
    while(true) {
      try {
        long start = System.currentTimeMillis();

        Item item = WorkQueue.getNextItem();
❷       log.fine("Processing item: " + item);
        item.process();

        long timeInMillis = System.currentTimeMillis() - start;
❸       log.info("Processing " + item + " took " + timeInMillis + "ms");
      } catch(Exception e) {
❹       log.severe("Unhandled exception: " + e);
      }
    }
  }
}
```

在❶处，我们创建了一个 Logger 实例，并把类名作为参数传递给它。这不仅可以自动地用类名来注解我们的日志信息，而且可以使我们把这里生成的日志信息与其他地方生成的日志信息区分开来。在❷❸❹处，我们以不同的日志级别（分别是良好级、信息级和服务级）生成日志信息，到底输出哪个级别的信息取决于我们如何进行配置——也许正常情况下我们只需要输出警告（WARNING）及以上级别的信息，但是在调试一个程序的时候，是不是应该把日志级别降低到最细微（FINEST）级别?

我们一直在讨论如何在准确地识别输入值以重现问题的情况下使用日志，但其实日志也可以用在更广泛的环境中，请看下面的讨论。


小乔爱问……

我应该把日志留在代码中吗

有些议题在开发者之间一定会引起争论，日志就是其中之一。

如果你为了进行问题跟踪，已经在代码中添加了日志功能，那么就应该把它放到合适的地方，以便以后问题再次发生时能很快地找到它。如果你使用的是可以很容易启用和禁用日志功能的日志框架，那么这么做就更正确了。这样不好吗？

那么，为什么会有争论呢？批评者会告诉你以下几点。

- 日志会使代码变得难以理解，使你只见树木不见森林。
- 日志可能会遇到与注释一样的问题——日志代码并不随着代码的更新而更新，这意味着你不能相信日志记录的内容，这会使情况变得比不用日志更糟。
- 无论你添加多少日志功能，它都是你不需要的。下一次你在同样的地方进行调试，还必须添加更多的日志。如果完成工作之后还把日志功能留在代码中，那么你只会使前两个问题变得更加复杂。

对于此类问题的大多数争论，我们的答案是要注重实效。日志是一种很有用的工具，但它可能被过度使用。如果你认为它确实有价值，可以考虑一直使用日志功能，但必须经过相关的训练。请确保你的日志是随时更新的，与代码保持一致，并且不会为了日志而做日志。

一般来说，最有用的日志是在最高（战略）级——记录发生的一切，例如由HTTP服务器生成的访问日志。低级别的、更战术性的日志不一定有长期价值，因此要确切知道添加日志之后它会给你带来什么。

如果发现日志会妨碍你的工作，但又不想失去使用它的好处，你可能应该了解一下面向方面的程序设计，它会提供给你将日志从主代码中分离的方法（一本好的参考书是 *AspectJ in Action*（Lad03））。

即将引爆的定时炸弹

当我写这一章的时候，部署我们的应用程序的服务器群集出现了一个硬件故障——一个SAN（Storage Area Network，存储区域网）系统突然将所有驱动器都标识出现了故障。我们可以十分肯定不可能所有的驱动器同时出故障，因此显然SAN系统本

身出了问题。

令人欣慰的是，出现问题的系统存有日志记录，供应商可以使用它来确认一个每49.7天就出现一次的时间窗口。利用3天内的输出信息，他们诊断出了故障并打了补丁。如果没有日志记录，他们就会面对一个莫名其妙的故障，必须花费大量的时间去尝试着重现这个问题（至少需要49天，直到那个窗口再次打开为止，甚至可能需要更长的时间，因为无法保证故障一定会发生）。通过抓取系统接收的输入值的关键信息和其内部的状态，他们可以大大缩短整个流程，在我们的系统再次出现问题之前就实施和安装修复程序。

2. 外部日志

把日志功能直接添加到软件中不是你唯一的选择，还可以通过拦截你的软件和其他软件间的流量，从软件外部获取很多有用的信息。

例如，你的软件通过网络与另一个软件系统通信，你可以在两个系统之间插入一个日志代理，如图2-1所示。如果日志代理不支持你正在使用的协议，或者你无法找到配置方法来使日志代理拦截流量，那么可以考虑使用网络分析器来捕获所有网络流量。你可以在A.4节中找到这些工具的使用方法。

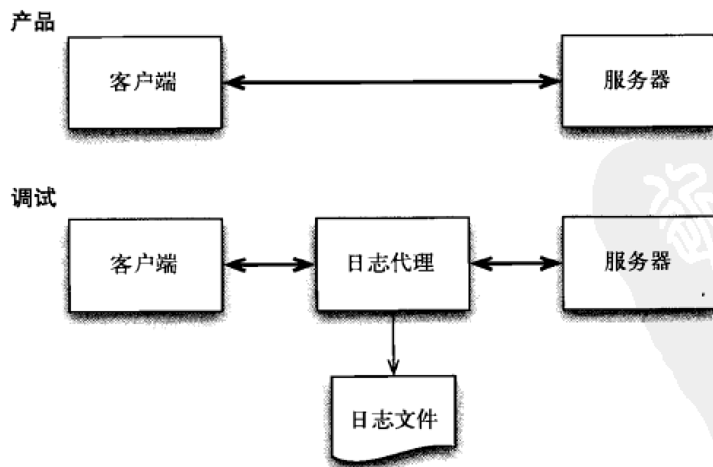


图2-1 日志代理

这种方法并不仅限于网络通信。如果你的软件通过API与一个第三方库文件通信，你也许能

够创建一个位于你的软件和库文件之间的中间程序（shim），[1] 来拦截它们之间的通信。这个中间程序能够链接到库文件并输出相同的 API，在记录日志时能够逐条地转发所有的调用。

你可能还发现，正在集成的系统已经提供了足够多的支持。例如，你正在编写一个Web应用程序，而应用服务器已经实现了细致又全面的日志功能。

2.4.3 负载和压力

有一些缺陷只有当软件在某种压力下运行时才能表现出来。这可能是因为软件本身也必须做一些事情（比如，处理大量同时发出的请求或特别大的数据集），也可能因为运行环境中的某些因素引起的（例如高负载的网络流量或有限的可用内存）。

由于一些显而易见的原因，我们很难重现这种负载对这类缺陷进行调试——因为我们很少能够拥有含成千上万测试人员的测试部门，去严阵以待地重现高负载的软件应用。

一个负载测试工具会执行一段脚本，这段脚本可以或多或少地模拟真实的应用。你可以通过调整配置，让它按照你的需要来创建尽可能多的并发会话（如果一个客户端不够用，那么可能会同时运行在多个客户端机器上），去重现任何你需要的负载级别[2]。

关于负载测试工具的问题，通常是找到一个办法让它们重现真实的负载。创建大量简单的交互行为是非常容易的，但是那样可能并不会产生足够真实的负载来重现你要调试的问题。解决的方法之一是使用日志记录真实的负载量，然后使用负载测试工具去重现它。

压力测试工具也是类似的，只不过它们不直接产生负载。例如，在软件的运行过程中你可以使用一个压力测试工具来分配和收回大量的内存资源，或者大量地占用 CPU 的运行时间。

你可以在 A.4 节中找到一些热门的负载测试工具的使用方法。

重现问题曾经是一个重要的障碍——毫无疑问，你现在正在跟踪一个真正的缺陷，你已经向诊断之路迈出了非常重要的一步。但是有一些有用的或者还算有用的方法去重现缺陷。在下一节，我们将看看如何改进你的问题重现，使之尽可能地有效。

---

① 在工程学中，中间程序是一块用来填充不同物体之间空隙的薄片。在计算机领域我们借用这个词来表示一个位于大的库文件和它的客户端代码之间的小的库文件。它可用于将一个 API 转换成另一个 API，或者像我们在这里讨论的那样，只需增加少量的功能而不必修改主库文件本身。

② 最近可供使用的云计算平台中，亚马逊的弹性计算云（EC2）平台可能是最有名的，它允许大量的客户端进行负载和压力测试，比以前方便很多。

2.5 改进问题重现

不管是什么样的重现问题的方法，只要有就比没有强。但是你想让重现问题既可靠又方便。在诊断期间你将不得不一遍又一遍地反复地进行问题的重现，所以你需要能够按需去做，并且代价最小。

2.5.1 最小化反馈周期

当运行实验来追踪你要调试的错误时，一个重要的原则是这些实验要尽可能地有效。如果实现完全可靠的问题重现需要运行 1 小时以上，或者需要你按照正确的顺序执行 50 个不同的动作，那么这样做也是效率低下的。

你要达到的目标是使你创建的“编辑–编译–执行–重现问题”的周期最短，错误最少。你希望能够快速地运行大量的实验，让你可以尽可能完全了解问题的所有方面（并最终测试可能的解决方法）。

> 你希望能够快速地运行大量的实验。
>
> *You want to be able to run lots of experiments quickly.*

和软件开发的其他众多领域一样，问题重现也是要使反馈周期最小。所经过的周期越短，反馈就越及时，其相关性也越高。

如果周期较长，可能会面临一个真正的风险，这就是你会发现自己得要一次做出多个修改——当讨论问题诊断的时候我们会看到，多个同步的变化将会导致各种问题。

1. 尽可能简单

> 要将问题重现最小化。
>
> *Aim for a minimal reproduction.*

第一次重现问题就能做到最小化是不太可能的。换句话说，相比所需的精简要求，它可能要复杂得多，因此你最先要关注的就是找出问题重现中哪些方面是不必要的，并把它们剔除掉。

例如，你的软件读取 XML 文件，而且你已经可以确定该软件读取一个包含 100 个标签的文件时就会崩溃。那么有一个很大的可能性，就是你不需要读取整个文件就能重现该问题。如果它在读取到一个特定的标签时崩溃，那么是否只需要一个包含这个特定标签的文件就可以了？或者在它周围仅仅需要多几个标签就可以了？

它可能没有那么简单——有可能是位于文件前面的东西创建了适合的上下文或那个标签，从而导致了缺陷的发生。然而，你可能会发现大部分的文件内容可以不用考虑。

问题重现中的哪些元素可以被剔除掉？往往多相信直觉。你了解软件，并且知道哪些模块可能被一些特定的输入所影响，哪些不会被影响。如果你的直觉不对，那么一些非直接的方法可能会产生奇效。

假设你有一个包含100行代码的输入文件，而且你并不清楚文件中的哪一行出了问题。那么你可以使用一种很简单的方法，就是先把文件的后半部分删除，看它是否仍然能够重现问题。如果是，显然你就可以把问题定位在文件的上半部分；如果不是，请删除文件的前半部分，你可能会发现文件的后半部分还存在缺陷。这样反复若干次后，你就可以迅速地将文件压缩成只有几行代码。同样的方法可以适用于任何类型的输入（通过用户界面进行的操作，从硬件得到的响应，等等）。

自动地最小化输入

事实证明，重现问题中让需要的输入值最少，是可以通过自动化的方式进行的。Andreas Zeller 在《代码之美》这本书中讨论了一种可以达到这个目标的方法（通过自动化二分法）。

就我个人而言，我从未看到这种方法在手工状态下使用，但是它确实是一种非常智慧的方法。也许它指明了一个工具支持的未来发展方向？

这种方法是二分法的一个特例，是在大规模的调试场景中非常有效的一种查询算法。我们将在3.2节中进一步讨论这个问题。

年轻的活力

在博士研究期间，我有幸能够在微软公司的编译器和工具小组做夏季实习。我的工作是开发代码查看调试器，并在此过程中发现了一个未发行的C编译器中存在缺陷。

我认为这是很有责任心并且非常有帮助的事，于是提交了一份缺陷报告，其中包括完整的源文件的预输出值（处理完所有的`#include`指令之后就已经有几千行输出了）。

一个多星期后，案子了结了，该缺陷已经有人提交过。研究这个缺陷的开发者给我发了一个简短的信息，说他已经将我的数千行代码压缩到关键的十行，很显然这只是重复了一个在几个星期前就被报告了的缺陷。如果我再多花一些精力来缩减我的报告范

围，我就可以发现这个缺陷是重复提交了，可以节省本来就很忙的同事大量的时间。

如果你不能找到一种最大限度地减少重现问题工作量的方法也不要太沮丧。有的时候真的是无法做到，而且在其他场合即使能够进行简化，在你能够简化之前也需要对问题有深入的了解。我们稍后会讨论的，改进你的问题重现不是一蹴而就的事，而是在整个诊断过程中要牢记在心的东西。

2. 最大限度地减少所需的时间

有些缺陷只是需要时间来重现——这不取决于你做了多少，而取决于你做了多长时间。比如某个Web应用程序在处理几千个请求后就会崩溃。这往往是某种类型的资源泄漏问题（内存、文件句柄或其他类似的问题）。

如果你怀疑这可能是出了问题，你有可能采取几种办法来使之早些发生。最明显的是，你可以在任何将用完的资源上做文章，要么采用直接的方式，要么通过修改代码在启动的时候占用大量资源，从而在正常的运行时能很快耗尽。此外，你可以假装营造出资源耗尽的局面，比如用一个适时失败的函数取代分配资源的函数，从而尽早再现问题。

2.5.2 将不确定的缺陷变为确定的

软件之美的一部分体现在它的确定性上——计算机精确地做着你让它做的事，而且如果起点相同，那么它每次都会精确地做同样的事情。然而，每个开发人员不论开发了多长时间的软件，都会遇到不确定性的软件——你每次做法一样，它却表现不一，忽左忽右。

> 不确定性只能有几个原因。
> *Nondeterminism can have only a few causes.*

那么，这种不确定性从何而来？这当然不会是随机的宇宙射线引起内存中比特值发生改变（不管你听说过多少老程序员的传说）。不确定性只能有几个原因：

- 开始于不可预知的初始状态；
- 与外部系统进行交互；
- 故意地使用随机性；
- 多线程。

我们依次考虑这些原因。


小乔爱问……

为什么不确定的缺陷是一个问题

设想一下，你正在处理一个每隔一次才能重现的缺陷。你认为自己已经进行了修复，但是因为你的重现是时断时续的，因此就不能简单地测试并判断说缺陷不会出现了，因为很有可能只是在这一次缺陷没有发生。每次测试成功之后，你都增强了信心，但是你永远不能完全确认已经修复了缺陷。

如果说判断是否已经修复一个间歇性的缺陷是很困难的，那么诊断这样的缺陷就更加困难。每次运行一个实验，你都不能确定观察到的运行结果到底是失败的还是成功的。这样是很难取得进展的，也极易陷入困惑，形成错误的推断，得到错误的结论。最重要的是，它令人十分沮丧！

1. 开始于不可预知的初始状态

当你的软件从未经初始化的内存读取数据时，通常就会出问题。现代的操作系统总是在你可以使用内存之前就对它进行了初始化，现代的编程语言不可能使用没有经过初始化的内存，这意味着这类不确定性问题较之以前没那么严重了。然而，C/C++程序员在某些环境中运行程序仍不得不担心这一点。即使你的程序代码是用 Java 编写的，你也可能发现与自己连接的第三方系统有这个问题，因此，你不能完全忽视这个问题。

如果你有理由相信，是这个原因导致了不确定性问题，那么你最好的选择可能是使用调试内存分配器（见 A.3 节），来强制内存被初始化为一个众所周知的值，或用内存完整性检验软件（见 A.4 节）来检测是否引用了未初始化的内存。

2. 与外部系统进行交互

与外部系统交互引起的不确定性问题往往不是因为二者表现不一，而是因为时间上微妙的不同。因为它在你的软件中并没有以锁定模式运行，当输入到来时，你的软件有时处于一种状态，有时处于另一种状态中。

如果你面对的是这种问题，解决的策略就是能够精确控制从外部系统接收了什么，以及何时接收的。因此，你最好的选择可能不是试图直接控制外部系统，而是用你能控制的东西替换它，比如调试子系统或代替测试（我们将在 9.1 节讨论代替测试）。

3. 故意使用随机性

随机性构成了一些软件的内在元素，例如，牌类游戏或者提供随机密钥的安全性软件。一些故意结合随机性的软件，其运行具有不确定性是没有什么好惊讶的。

幸运的是，大部分所谓使用随机数的软件并不是真正意义的随机，而是通过确定性算法产生的伪随机数，这种做法在随机性上做得很好。他们有个非常有用的属性，那就是如果你设置一个已知值作为种子（作为初始化随机数生成器的值），那么你将会从随机数生成器中得到同样的序列号（因此，这是个完全可以预测的行为）。

4. 多线程

由多线程引起的不确定性尤其难以处理。在单核 CPU 中，一个线程可以在任何时候中断另一个线程。在多核系统盛行的今天，往往我们处理的并不是真正的并发。

如果有可能，你可以仍用原来的方法重现问题，最简单的解决方法常常是运行软件而不用任何线程。如果不能这样，那么你必须考虑使软件能够在你的控制下随环境变换，而不是让它听调度程序随机摆布。这种方法简单与否，依赖于软件的设计，以及你是否构建了这样的控制。

在缺乏并发控制的结构化方法的情况下，你将不得不依靠一些特殊的方法。因此，在你的处理方法中最有效的工具之一就是不起眼的 `sleep()`方法，这个方法允许你强制一个线程长时间等待而出现竞争状态（这是一种依赖于事件的序列与时间的行为）。

例如，假设你正在工作的软件中多个工作线程并行处理工作项目（多线程软件中一个普通的模式）。工作线程使用下面的 Java 代码来获得工作项目：

```
if(item = workQueue.lockWorkItem()) {
  item.process();
  workQueue.writeResultAndUnlock(item);
}
```

你在试图跟踪一个间歇出现的缺陷：有时同一个工作项目会同时分配给两个工作线程。遗憾的是，这种情况极少出现。你可以把代码修改如下来增加重现这种问题的几率。

```
    if(item = workQueue.lockWorkItem()) {
❶     Thread.sleep(1000);
      item.process();
      workQueue.writeResultAndUnlock(item);
    }
```

在❶处调用 `sleep()`方法可以大大增加出现竞争状态的可能性，使问题更容易出现。

注意，尽管 `sleep()`方法在重现问题和诊断阶段很有用，但在修复缺陷阶段它不是一个合适的方法。我们将在 8.3 节作更深入的探讨。

2.5.3 自动化

对一些步骤运用必要的自动化手段来重现缺陷，不但可以加快进程还可以减少犯错误的几率。越是复杂的重现，运用自动化受益就越多，但是对于一些非常简单的测试要考虑是否值得这么做。

1. 自动化测试

可能最有效的方法就是使用你的自动化测试框架（假设你拥有一个）。一个自定义测试不仅能够方便地运行，而且当诊断结束开始修复工作的时候，对于即将编写完成的测试来说，它是一个很好的起点。

另外，如果你的重现需要一长串用户接口行为，可能就要考虑使用 A.4 节中的用户接口测试工具。

2. 重放日志文件

如果你确定缺陷重现需要通过日志，那么你可以有另一种选择——重放日志文件。在图 2-1 中，我们说明了日志代理如何被用来记录你试图调试的软件与第三方服务器之间的交互。在图 2-2 中，我们可以看到从日志读取的第三方服务器的仿真版本，可以用来重新构造任意相同的操作序列。

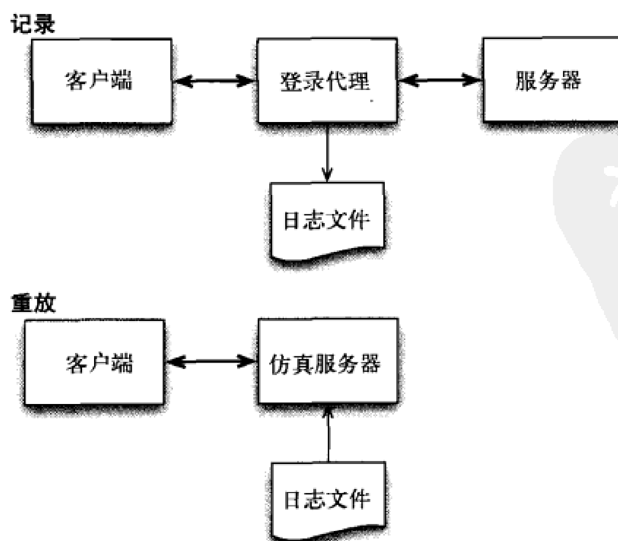


图 2-2 在日志文件中重放

2.5.4 迭代

在诊断过程中，你构建了越来越多的关于如何以及为什么软件如此运行的信息。你可以而且应该使用这些信息来不断地改进你的重现，如图 2-3 所示。

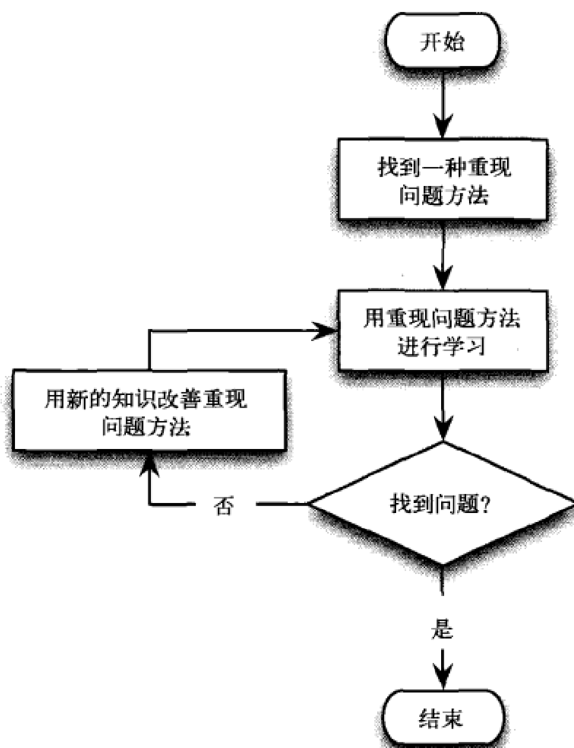


图 2-3 改进重现

比如，最初你通过给应用程序提供大的输入文件来重现问题。你一开始想尝试把文件变小，但没有成功，并且（甚至更差）运行三遍程序但缺陷仅出现一次，那么你可以按照如下步骤反复地改进你的重现。

(1) 你确定一个特定的模块已包含在内，其中有导致缺陷的元素。这样就可以创建一个更小的文件。

(2) 进一步诊断，发现你能通过用桩模块替代与第三方服务器交互的子系统，让问题每次都出现，桩模块可以很容易返回已经确定的响应。

(3) 最后，你把跟踪范围缩小到一个特定的函数，通过设置一组具体的参数调用该函数来创建单元测试，以便重现问题。

调试的艺术就在于总是像上面那样寻找机会使自己的生活更简单。

2.6 如果真的不能重现问题该怎么办

有的时候，无论你怎么努力，也无法重现你所跟踪的缺陷，那么该怎么办？

2.6.1 缺陷真的存在吗

当然，一种可能性就是你在追逐一个幻想，而实际上缺陷并不存在。如果你有足够的证据证明缺陷不存在，这当然很好。但是请小心，你必须确定已用尽了所有可用的方法——以我的经验看，软件开发者通常过于草率地得出“缺陷不存在”这个结论。

如果你决定以“需要更多的信息”或者“我看运行正常”（或者任何相似的状态）为结论，那么不要简简单单地就此终止。用户一般不会恶意地报告缺陷。对他们来说极有可能是软件真的运行出错了。可能他们没有非常清楚地解释为什么运行出错了，或者误解了软件的一些方面。花一些时间来向他们描述你所做的事情，想办法获取些额外信息，帮助你彻底弄清楚他们遇到的问题。

2.6.2 在相同的区域解决不同的问题

在你重现问题的同一区域还有其他缺陷吗？如果有，即使它们（从表面上看）没有你目前跟踪的缺陷那么严重和紧急，但也值得一查到底。有两方面的原因来说明为什么应该如此。

第一，这是清理这个区域代码的好方法。你可能会发现真正寻找的问题被其他问题掩盖了。将这些问题解决了，会发现原来要找的缺陷将更加清晰。

第二，解决一个可以重现的问题是从总体上更好地了解代码的好方法。这种深入的理解能帮你了解程序的内部机制，并找到重现问题的关键因素。

就算这些都没有帮助，最坏的情况是你将修复一些并不紧急的缺陷。

2.6.3 让其他人参与其中

开发人员很容易有盲点。我们必然与用户具有不同的视角，同时这也意味着我们将失去一些重要的信息，而这些信息从用户的视角来看可能显而易见。此外，我们的工作重点是让编写的软件可以正常运行，而不是证明这个软件有缺陷。

因此，让一些可以从别的角度解决问题的人参与进来是非常有帮助的。例如，你的客户服务

团队可能对你的用户非常了解，而你的测试团队之所以存在，就是为了找到各种方法来证明软件是存在缺陷的。

如果你可以和第一时间反馈错误的人进行交谈那是最好的。我们将在 6.2 节深入讨论这点。

2.6.4 充分利用用户群体

如果缺陷出现在外部系统中，而不是发生在你的开发系统中，或许应该让用户为你收集所需要的信息。但这种方式并不理想，理由如下。

- 你需要通过一些方法找到该软件的工具版本给用户。
- 它仅仅适用于一些用户群体——他们得准备不厌其烦地帮你解决一些问题，而且还需要具备相当的技术实力。
- 从你决定需要收集什么样的信息到从相关领域获得这些信息，这个迭代的周期可能远远超过你的预期。

但是如果所处的环境允许你这样考虑，它会是非常有效的。如果工作的平台是开源软件，这种方法就特别值得考虑，因为开源用户群体可以更加开放地参与到调试过程中。

2.6.5 推测法

虽然使用实证方法进行调试通常是最好的，但它不是唯一的解决办法。如果你不能重现缺陷，那么能够使用实证方法的一个重要工具就丢失了，这样你就必须寻找其他的解决方法。

其中一种方法就是纯逻辑推理，思考为什么软件的一些功能要这么做。这么做可能会非常费时和棘手。但是当其他方法无效时，它往往可以起到作用。

你现在需要做的就是“把你自己融入到软件中”，在你的想象中执行软件。执行到每一步时，考虑有哪些出现错误的可能性，尝试解释你跟踪的缺陷。

复制带来的问题

我们曾经开发了一个基于 MySQL 数据库的 Web 应用程序。MySQL 提供了一个非常有用的复制功能，我们用这个功能设置两台服务器，一台主服务器，一台从服务器。主服务器承担所有的工作，从服务器复制所有发生在主服务器上的工作。这意味着如果主服务器宕机，从服务器可以作为备用服务器使用。生活真美好。

大部分情况下是这样的。

但是偶尔几次，从服务器崩溃了，只提供了一些非常模糊的错误信息。唯一能使系

统复活的方法就是重新创建从服务器上的主服务器数据库的备份，从头开始配置复制。有些时候从服务器在重新复制几天后就崩溃了，而有些时候连续运行几个月都没有问题。

很明显出问题了，但是我们不知道问题出在哪里。我们排除了硬件出错的可能性，因为我们可以互换主从服务器，这么做仍然会出现从服务器刚才出现的问题。我们不可能在测试系统中复制这个问题——它只可能出现在产品服务器上，日志也没有提供给我们有用的信息，搞不清到底发生了什么。

我们担心，或许唯一的解决办法是要买两台以上的服务器用来重现问题，同时编写一个非常复杂的测试用来模拟现实的负载。很明显，追踪这种缺陷将会是长时间的，涉及面很大，十分昂贵。

然而在此之前，我决定仔细地回顾一下我们之前编写的用来复制的脚本。我把脚本打印出来，又打印了一份 MySQL 文档的副本，还有所有我通过 Google 能够找到的关于 MySQL 复制可能会出错的相关信息，坐下来仔细地研究。通过几天对脚本的分析，在白板上绘制图表，和小组其他成员讨论可能出现的问题（“我是主服务器，你们是从服务器，Thomas 是客户端——这将会发生什么……？”）。最终我们探测到了问题出在加锁时出现的一个竞争条件，我们用它来确保得到一个不间断的数据库快照。

确认问题之后，修复便很简单了，从此从服务器又可以顺利地复制了。

令人高兴的是，如上例的这些情况是非常少见的，正常来说你有能力重现这个问题。在下一章，我们将会看到如何使用重现来诊断问题。

2.7 付诸行动

- 在做任何事情之前找到问题重现的方法。
- 确认你运行的版本和提出缺陷的版本是相同的。
- 复制报告缺陷时使用的环境。
- 确定重现缺陷需要的输入数据：
  - 推测数据；
  - 通过日志记录合适的输入数据。
- 通过反复优化确保你重现问题的程序既方便又可靠。
  - 减少运行步骤、数据量或需要的时间。
  - 去除不确定性。
  - 自动化。



第3章 诊断

诊断问题是程序调试的关键。这个阶段我们可以开始解决缺陷问题了，你可以了解看到的运行结果背后的根本原因。

在这一章中，我们将介绍以下内容：

- 核心的诊断过程；
- 各类实验以及如何才能进行一次好的实验；
- 一些有用的策略。

3.1 不要急于动手——试试科学的方法

尽管你可以使用各种各样的工具和技术，并利用软件自身来帮助你找出缺陷，但是请记住，你最重要的财富是你的智慧。这一点永远都不会变。你是用你的脑子而不是用计算机来进行诊断。

> 兼顾创造性和严密性。
> Balance creativity with rigor.

当调试程序时你需要的思维方式和侦探破案、科学家调查某个新现象的思维方式是类似的（因为问题是类似的）。真正有效的缺陷修复要求思维方式既开放又有条不紊，解决方法既创新又注重全面综合，这和软件开发的其他很多方面是一样的，说白了就是寻找一种方法，能够平衡以上看似矛盾的需求。

科学的方法应该在正反两个方向都可行。[①]我们可以先提出一个假设，然后再通过实验去证明假设正确与否，实验结果要么支持假设，要么否定假设而取其对立面。我们也可以先提出一个与现实理论不符合的观察结论，进而去修改现有的理论或者干脆将错误的理论彻底修正。

在调试的过程中，我们几乎总是使用后者。观察（出现缺陷）证明我们的理论（软件应该像

① 学习科学历史与哲学的人会意识到我略过了许多细微之处。

我们想象的那样运行）是错误的，正如托马斯·赫胥黎所说："科学的最大悲剧在于美丽的猜想被丑陋的事实所扼杀。"

一种调试方法

当发现事情出乎意料之后，你的任务就是去修正你对软件的理解，直到你确实了解了它真正的运行状况。为了做到这一点，你需要按照第一种方法来做——提出一个可能提供解释的假设，然后再构建实验去证明你的假设。

因此，对我们来说比较理想的处理过程应该是这样的（见图 3-1）。

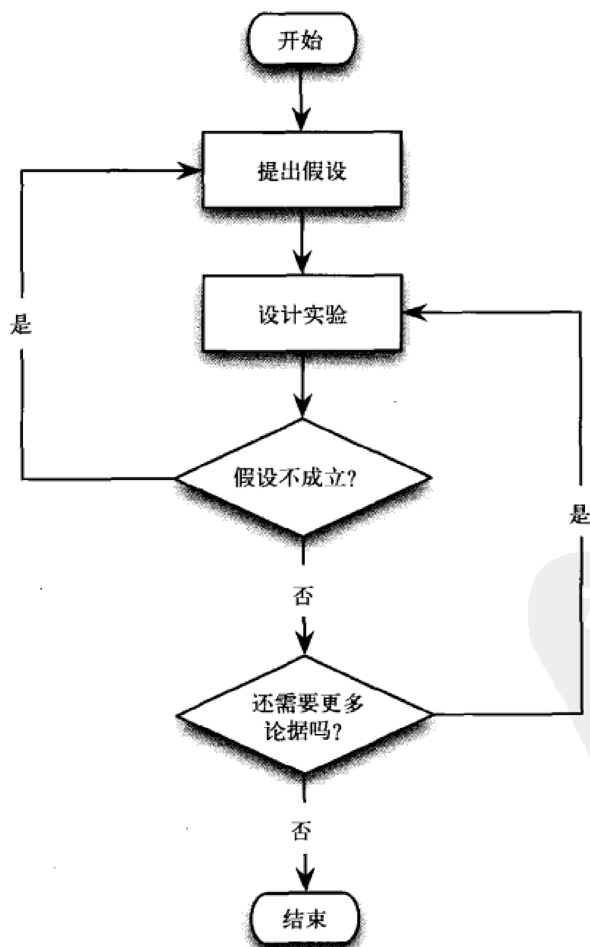


图 3-1 调试方法

(1) 按照你对软件运行情况的理解，提出一个可以导致这种运行状况的假设。

(2) 设计一个实验，证明你的假设正确与否。

(3) 如果你设计的实验不能证明你的假设，那么重新设计一个实验，然后再次进行实验。

(4) 如果实验支持你的假设，那么继续进行实验，直到能证明或证伪你的假设。

这种方法十分有效，但却十分抽象。怎样才能把它转化为实际行动呢？

1. 不同类型的实验

实验起点就是我们在第2章中已经详细阐述过的问题。由此出发可以进行几种不同类型的实验，每一个都会换个角度再现问题。

- 你可以检查该软件内部状态的某个方面（或者直接运行程序，或者利用调试器运行）。
- 你可以改变软件运行的某个方式（例如，改变输入的参数，或者换一个运行环境），看它的运行结果是否有所不同。
- 你可以改变软件本身编码的逻辑，检查这种变化的影响。

做出什么选择要由你的假设的性质而定，而能否做出最佳的选择取决于你的经验和直觉。

然而无论你选择哪一个，请记住，最重要的是你的实验必须要有一个明确的目标。

2. 实验必须起到验证的作用

实验是一种达到目的的手段，而不是目的本身。如果不能用实验证明一些东西，那么你的实验就是毫无意义的。

实验能告诉你什么呢？
What is your experiment going to tell you?

在花费时间和精力来构建和运行实验之前，请首先问问自己实验能揭示出什么。有哪些可能发生的结果？如果没有一个实验结果能够对你的诊断提供帮助，那么不如趁早换一个。不要以为你在行动就是取得了进展，如果你设计的实验无法加深你对软件的理解，那么这种实验只是在浪费时间而已。

你可以设计实验来证明或者推翻你的假设。虽然听起来似乎有悖常理，但事实上常常是推翻假设的实验更有用。可以说这是因为彻底证明一件事是很难的（因为你看到了你所期望看到的并不意味着它是按照你认可的原因产生的），但其实主要还是一个心理问题。

对于发生的事情，如果你已经有了一个似乎合理的解释，就很容易想当然地认为它们是正确

的。与他人争辩并试图推翻你的假设是非常有效的方法，这可以帮助你找到自己的想法中未曾意识到的漏洞。如果假设始终成立，尽了最大的努力也无法推翻它，那么你可以底气十足地宣称你的假设坚不可摧。有时你自己都会感到惊讶，发生的事情和你原来的想法大相径庭。

3. 每次只做一个修改

设计实验的基本规则之一是，你每次只能做一个修改。

> 多个修改会导致错误的结论。
>
> *Multiple changes lead to misleading conclusions.*

如果你仅仅做出一处修改就得到了结果，那么就基本可以确定是这个修改导致了结果的变化[①]。如果你做出的修改不止一处，那么我们就很难确定是哪处修改导致了哪个结果。这多处修改之间也可能会发生一些意想不到的相互作用。最乐观的结果是，通过实验你可能无法得到有用的信息。但最坏的情况则是，你得到的错误结论将你引向了一个完全错误的方向。

此规则适用于任何修改，源代码的修改、环境的变化、输入文件的变化等都适用。事实上，它适用于任何可能影响软件运行的要素。

但是令人大跌眼镜的是，由于某些原因，这个原则却经常被人忽略。我已记不清有多少次看见有些人一次做了多个修改，然后就根据实验结果得出结论。虽然同时做几处修改看起来比较节省时间，但实际上你所得到的可能只是无效结果。请牢记这一点，不要再犯这种错误。

最后，当你发现程序运行结果改变时，就撤销所有引起这种变化的操作，看看程序是否恢复到原来的状态。这一招很管用，能清清楚楚地表明你看到的就是原因和结果，而不是意外的发现。

4. 记录你所做过的调试

如果你连续几天甚至几个星期都在调试一个缺陷，那么最终你会进行很多不同的实验。理想情况下，每个实验都会消除掉一些可能的原因，并且最终你也将会找到根本原因。

当对软件进行长期的调试，做过很多此类的实验时，你很有可能会忘记自己已经做了哪些工作。这意味着你可能会在之前已经得到证明的实验上面浪费时间，或者进入一个死胡同。最坏的情况下，它可能导致你得不出结论，或者得到一些错误的结论。

① 但还不能完全肯定，一个不断变化的底层系统可能推翻这种解释，但基本上可以如此假设。

> 定期回顾你已经尝试过的实验和学到的东西。
> *Periodically review what you've already tried and learned.*

最好的对策就是把做过的实验和结果记录下来。不必花费大量时间在这方面，也不必做得特别细致，只要能够确保你不会忘记自己曾经做过什么就足够了。定期回顾所做的笔记可以巩固以前的知识，也有助于找出最有可能解决问题的后续步骤。

许多开发人员发现记日记是十分有帮助的。他们可以记录会议的内容、设计草图、安装某个软件的必要的步骤，事实上，任何将来有参考价值的东西都应该记录。日记簿能够详细地记录你所做的实验。如果希望保留电子笔记，你可以考虑使用个人的维基。

5. 不要忽略任何细节

有时候你可能遇到莫名其妙的现象。你运行一个实验，预期的结果是 A 或者 B，而实际结果却是 C。或者你编写了一组关于如何重现问题的指令集，但是软件运行的结果却出人意料。

这时你可能以为“这是不可避免的”，所以对它不予理睬，着手开始采取其他的策略。千万不要这样！软件正试图告诉你一些事情，为了自己的利益你应该去听听软件想对你说什么。

发生了意外意味着你所做的一些假设是不正确的。可能是关于软件如何运行的假设，可能是关于你正在跟踪的缺陷的假设，可能是你已经构建的实验的假设，诸如此类。如果你的假设存在问题，那么你能做的最有价值的事情就是停下来，确认问题所在，然后修复问题。如果不这么做，就将前功尽弃，得出的所有结论都不可信。

塞翁失马，焉知非福，只有这样才能了解软件真实运行状况。弄清楚意想不到的运行状况可以为你节省大量的跟踪缺陷的时间。

> 凡是你不明白的都是潜在的缺陷。
> *Anything that you don't understand is potentially a bug.*

即使你发现的这些莫名其妙的现象和你手头工作无关，你的发现本身也往往是非常有价值的。凡是你不明白的都是潜在的缺陷。如果能证明它与你正在做的工作并不相关，你可以把它放在一边，但不要忘记它。要把它们记下来（或许可以生成一个缺陷文件），经常重新回顾一下。往往就是你无意间发现的这些东西后来恰恰被证明是需要修复的缺陷。你肯定宁愿自己修复以这种方式发现的缺陷，也不愿意等到愤怒的客户把缺陷报告给你。



小乔爱问……

调试的时候日记簿还有其他什么用处吗

除了保存你的实验记录，日记簿还有如下的好处。

- 写出假设。将曾做过的假设记录在纸上，可以帮你分析出假设的缺陷，尤其当假设很复杂的时候。
- 可以跟踪详细信息，如栈跟踪、参数值和变量名。这不仅可以帮助你再发现一些问题，而且在解释这个问题时，它也可以帮助你与同事进行沟通。好记性不如烂笔头，单纯依靠记忆是不可靠的。
- 把你想要尝试的想法列成一个清单。你可能会经常注意到其他一些想要调查的事物，或者你想到一个接下来需要做的实验，但你又不想放弃目前的实验。写一个“任务”列表可以确保你不会忘记要做的事情。
- 当你短暂休息时，不妨在上面随意涂鸦吧。

不可忽视鬼祟现象

我查阅了昨天的服务器日志文件，收集能帮助我诊断当前某个问题的证据。不经意间，我注意到一个客户好像是遇到了连接问题——他不断地退出登录又重新登录。

这和我跟踪的问题毫无关系，我完全可以置之不顾。毕竟连接出现问题是很正常的。但我还是感觉不对劲——他的登录模式太有规律了。我强烈预感到这里面一定有鬼。

果然，这位有问题的用户发现了一个不为人知的方法，可以越过软件实施的安全机制（这个软件可以给每一位用户分配其可以使用的特定资源）。通过退出登录又立即重新登录这种方法，他可以重设给他的配额。我们现在既然发现了这个问题，也就自然能够很容易地解决它。

3.2 相关策略

虽然每个缺陷都是不同的，但以往的经验已经多次证明某些技术和方法在追踪大量问题方面很有效。它们不一定能解决你遇到的每一个问题，但每一个程序员都应该熟知这样的技术和方法。

3.2.1 插桩

诊断的所有内容都是与信息有关的，准确估测系统的状态、程序的执行路径等。虽然可以通过很多途径推测或得到这种信息，但是到目前为止，最简单、最直接的办法就是在你的软件中使用插桩技术。

插桩技术本身也是代码，它并不影响软件本身的运行，但是它却能告诉我们为什么软件会这样运行。在前面的章节中我们已经介绍了最常用和最重要的注释类型——日志。或许最早的调试技术就是在代码[①]中加入临时的日志文件来，肯定或否定我们对于软件运行状况的理解。

随时利用语言环境。
The full facilities of the language are at your disposal.

但是插桩不仅仅限于简单的输出语句，你可以充分利用语言环境。你可以收集和整理数据，评估任何代码，测试相关条件，总之，没有做不到的，只有想不到的。

注意海森堡定律

量子物理告诉我们，对系统的观察可能导致系统本身的变化。计算机软件不是量子力学（至少现在还不是），但我们仍然需要保持警惕。

对软件进行插桩实际上会改变这个系统。这可不是简单的观察，它会增加影响系统行为的因素。在诊断分析的过程中，这样做是危险的，因为在进行实验的同时引入无目的性的改变很可能导致你得出错误的结论。

从根本上讲，我们没有办法保证不引入一些副作用。你自己修改源代码意味着内存中目标代码的布局和它执行的时间将会受到影响。令人欣慰的是，多数情况下这只是我们瞎担心而已，只要小心避免更多明显的副作用，这个问题通常都是可以忽略的。

不过，最好还是让源代码尽可能接近其原始形式。不要让失败的实验及其可能带来的副作用随时间而累积。保持其原始状态也有助于确保代码易于理解（或至少没有更难），并确保你在最终着手修复问题的时候不要引入目的不明的改变。

让我们来看一个例子。假设你正在追踪 Java 代码中数据结构方面的错误，并依次处理各个节点：

① 通常就是以C语言的同名函数命名的 printf()函数调试。

```
while(node != null) {
  node.process();
  node = node.getNext();
}
```

会有提示说，有节点被处理了多次。(换句话说，get Next()不止一次返回一个或多个节点。)但是，我们并不知道不止一次返回的是哪些节点。

可以使用插桩技术解决这个问题：

```
❶ HashSet processed = new HashSet();

  while(node != null) {
❷   if(!processed.add(node)) {
      System.out.println("The problem node is: " + node);
    }

    node.process();
    node = node.getNext();
  }
```

在标识❶处我们创建了 HashSet，可用它来存储已经处理过的节点。在标识❷处，当前节点被添加进集合中。如果该节点已经在集合中了，add()函数会返回 false，表明此节点已得到处理。

通常情况下，我们都会像这样迫不及待地使用插桩技术，并在取得预期效果后立即停用。但实际上插桩不应该是这种临时抱来的佛脚，完全有理由把它留在代码中，从而编写出自调试软件。在 10.1 节我们再介绍相关方法。

3.2.2 分而治之

分而治之，或者叫二分法，可谓是缺陷调试的一把瑞士军刀，在各种不同的情况下它会一次又一次地出现，帮你解决难啃的骨头。

二分法是一种搜索策略，比如说你有一个包含一百万个整数的分类数组，你试图确认某一个数是否在这一数组中。你可以不动脑筋地依次检查每个数，但平均下来你大概需要检查五十万次才可找到你想要找的那个数。在最坏的情况下，你可能需要检查一百万次。

或者，你可以找到该数组的中间点（把数组划分成两半，每部分有 50 万个数字）。如果该数字大于前半段的最后一个数，你只需要在下半段寻找即可。如果不是，那么你只需要在前半段数组中查找。选择你要找的那半段，然后依此方法再分（现在是每段 25 万个数）。如此继续，那么在 20 步后你肯定会找到你要的结果（通常，二分法需要的步骤不多于 $\log_2 N$，N 为搜索的条目数）。

该过程的最后几步见图3-2。

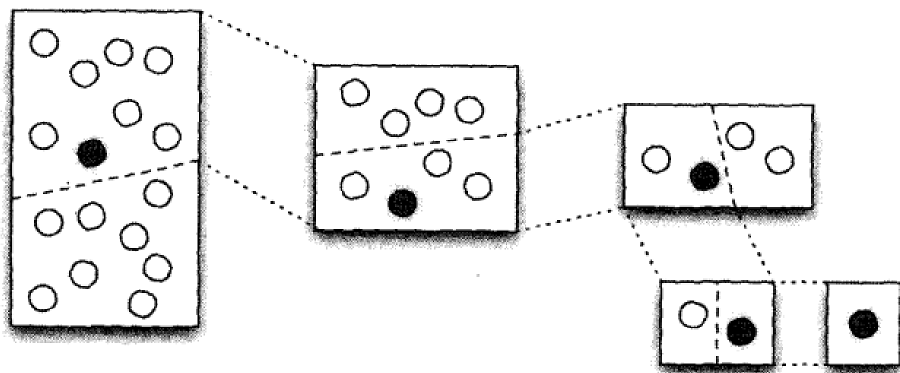


图3-2　二分法

我们在2.5节已经讲过一个例子，用二分法帮助我们有效地调试了程序，当然还有许多其他情况。

你可能在跟踪内存损坏的情况，你有办法检测到这一情况（可能在损坏之后，应该为零的变量已不复为零），但是你无法知道在数千行的代码中到底是哪几行代码导致了这种状况。将检查点插入到程序代码的中间处，然后执行。如果这时候同样的破坏情况还是发生了，那么你可以判定，是代码的前半部分发生了错误；反之，一定是代码的后半部分发生了错误。如此重复进行下去，你很快便会发现错误的代码行所在的位置。

有的时候这种方法并不是一劳永逸的，但用这种方法至少可以帮助你排除很多错误的情况，提高效率。如果你的程序有若干个相互独立的模块，那么尝试禁用所有这些模块，看同样的错误是不是又发生了。如果又发生了，那么你就可以排除掉这些禁用的模块，不用再对其进行测试了。如果没有发生，那么你可以继续使用二分法排除。最终你要做的只是对某一个模块进行查错，这对减少工作量来说无疑有相当大的帮助。

但是不要太依赖于这种方法，只有当你的搜索空间可以被分成均等的两段时，这种方法才是最有效的。我们要的是减少工作量、提高效率的方法，这种方法并非只有二分法一个。

在下一节，我们将讨论如何用源代码控制系统找到回归错误。你猜是什么方法？就是二分法的另一个应用实例。

3.2.3　利用源代码控制工具

有时你会发现陷入了一种回归状态：一个缺陷本来不影响正常运行，但做了改变后却变成了

实实在在的缺陷了。对于此类缺陷，同样可以使用常用的诊断方法，但是在回归跟踪时有一个特别有价值的工具——源代码控制系统。

如果你能准确地识别是哪个变化导致了这些问题，那么诊断其发生的原因就轻而易举，你的源代码管理系统记录了对程序进行的所有修改，你所要做的就是从中揪出导致该问题的罪魁祸首。

第一步，你要重新检查所有的签入注释——可能罪魁祸首已经很明显了。但是，如果不是，那么你可以使用以下的步骤来准确地定位所做的修改。

假如你已知错误不是在版本 2.3 中，而是在当前的版本 3.0 中，而从 2.3 到版本 3.0 进行了 200 多次签入。现在就可以按常规处理了。签出并创建一个中间版本，看看缺陷是否还存在，如果没有，那么一定是此后更新的版本中出现了问题；如果有，则说明问题是在较早的版本中引入的。如此反复多次之后，你最终就会找到导致该问题的原因。[①]

有时你怀疑某处更改有问题，但却想不出为什么会出问题。可关键不在于找到了导致问题的更改，而在于通过调查排除了更多代码的嫌疑。

3.2.4 聚焦差异

你的软件正常情况下都是能运行的。经常是不管是谁来操作，要诊断的缺陷可能都不会对正常运行有影响。所以，你要寻找的是一些特例或特殊用户。你已经知道它只影响这种情况，你需要做的就是找出它是什么。

通常，当你尝试重现问题的时候这些差异就会显露出来。问题是否只发生在一个特定的环境下？那样的话，问题最可能存在于针对该环境的代码中。问题是否只发生在使用大量输入文件的时候？那样的话，最有可能出的问题就是资源泄漏或超限问题。

如果在重现问题阶段这些差异没有显露出来，那么“找到缺陷的边界”可能非常有用。如果你能找到几个类似的软件运行的方式，其中有一些重现了问题，但有些没有，那你就要当心了。

3.2.5 向他人学习

许多缺陷只在你的代码中发生，因此只有你及合作者才能解决它们。但有时缺陷将涉及广泛使用的技术（例如你的编译器、库文件或你使用的框架），在这种情况下，很有可能其他人在你

---

① 这种方法非常有用，Git 源代码控制系统都以 `git bisece` 命令的形式为它提供直接的支持。更多细节请参见 *Pragmatic Version Control UsingGit* [Swi08]。

之前就已经遇到同样的问题了。

在这种情况下，先在网络上搜索一下可以事半功倍。也许有人在论坛上问过同一类错误模块的问题，或写了一篇博客痛陈自己犯下的错误，而这正好也是你想搞懂的问题。

3.2.6 奥卡姆的剃刀

> 其他条件相同的情况下，最简单的解释是最好的。
>
> *All other things being equal, the simplest explanation is the best.*

人们常说“奥卡姆的剃刀”，其意思就是：“其他条件相同的情况下，最简单的解释是最好的。”

这只不过是一个经验法则——符合事实的多种解释可能都是对的，不管它多么令人难以理解、多么错综复杂甚至不合情理。但是你必须先选择一个来探究其真实性，通常最简单的解释是最好的。

3.3 调试器

调试器既有简单的命令行调试器，也有完全集成在 IDE 中的调试器，其复杂程度和处理能力千差万别。它们的共同之处就是可以在代码运行的时候对代码进行检验、设置断点、单步调试、检查程序运行状态。

这好像有点奇怪，为何我现在才讨论它呢？对于一些开发人员来说，调试离不开调试器，它是他们使用的最重要的，还可能是唯一的工具。

毫无疑问，调试器是工具箱中最强大的一个工具，你当然应该花时间去熟悉它，了解能做什么，擅长做什么。但这里要说明的是：随着时间的推移，我发现自己使用调试器的时间越来越少。而且这不是我个人的感觉，许多其他开发人员在交谈中告诉我，他们也与我的感受相同。那么，这是什么原因呢？

带来这种变化的就是测试优先的开发思想（见 9.1 节）。以前我遇到问题时的第一个想法就是使用调试器，但是现在我首先做的就是编写一个测试程序。要理解我为什么这样做，就要思考一下为什么要使用调试器。在软件开发生命周期的如下三个不同阶段，调试器都特别有用。

(1) 在开发过程的初期，调试器是非常有帮助的，对代码进行单步调试有助于使我们确信软件运行的结果和我们想要实现的是一致的。

(2) 如果我们想让代码以一种特定的方式运行，就可以使用调试器来确认或反驳这个想法。

(3) 最后，调试器可以帮助我们探究看不懂的代码。

但是有了测试优先开发这个流程，形势就大不同了。现在，不必逐步跟踪代码来检验程序是否按照我们预想的那样运行，而是写一个或多个测试程序来证明它确实按照我们预想的那样运行了。如果对于缺陷原因有设想，我们就可以创建一个测试程序来证明这一点。这样做的好处在于测试结果可以永久保存。这就与在调试器里逐步跟踪代码不同，调试器的结果是短暂的。测试不仅现在证明了代码是工作的，而且今后仍然能证明，还能被其他的团队成员运行（甚至是改善）。我们不仅用测试证明了理论的正确，而且随后还可以用它来证明我们的修复程序解决了问题。

> **调试会话是短暂的，测试是永久性的。**
>
> *Debugging sessions are ephemeral; tests are permanent.*

交互式控制台

如果你使用的是解释性语言，比如 Python 或 Ruby，那么你可以使用另一种有效的工具——交互式控制台。这可以让你直接输入语句并让它们立即执行，甚至如果你愿意的话，可以重新定义功能。控制台是一个非常有用的研究工具，无论是在调试时，还是在尝试新的东西观察其如何工作的时候。

如果你使用的是编译语言，也还是可以使用交互式控制台的。一些供编译语言使用的更复杂的调试器可以提供一些非常接近于交互式控制台的工具。这不是完全一样的东西，但非常相似。

所以，可以把调试器作为一个探索工具。可以肯定的是，调试器非常重要，但是它起到的作用比前些年小了很多。

顺便说一下，如果你使用一个非常新的开发环境，比如 Ruby，用调试器就十分方便。说得委婉一点儿，目前的 Ruby 调试器很“原始”，但是相对于几年前来说问题少了很多，因为现在调试器用得少了。

3.4 陷阱

在诊断期间有无数的方法会误导人，但是反复出现的方法并不算多。在本节中，我们将看看从实战中获得的一些来之不易的经验教训。

3.4.1 你做的修改是正确的吗

如果你做的修改似乎没有任何效果，那么你并没有改到点子上。也许你编辑的是某源代码树

> **如果你做的修改似乎没有任何效果，那么你并没有改到点子上。**
>
> *If the changes you're making have no effect, you're not changing what you think you are.*

上的文件但是编译的却是不同的文件，或者你编译了正确的文件，但却运行了错误的可执行文件，或者你编写的代码被预处理器禁用了，或者你的浏览器本应访问开发服务器却访问了产品服务器……诸如此类。

这些陷阱太常见了，很容易就会掉进去，又让人摸不着头脑（直到你突发灵感的时候，你才突然意识到自己的错误），使你备受折磨。唯一的防御措施就是在潜意识里要时刻提高警惕。

承认自己已经掉入陷阱的最简单方法就是，在代码中特意引入很明显的失败。如果你在使用C++，或许就是一个明显的语法错误或`#error`指令？或者调用了`System.exit()`？当编译过程不能正常停止或应用程序运行得十分勉强时，就应主动查找你的（现在是显而易见的）错误。

3.4.2 验证假设

你所做的一切都是基于假设的。你不可能不做假设，不做假设是疯狂的尝试，因为你不可能每一次都从最基本的原理开始工作。

但假设是危险的，因为它们会带来盲点，即没有必需的直接证据时就断定这些假设是正确的。

有些假设的危险性不大。例如，假设你的编译器忠实地将源代码翻译成正确的目标代码应该是安全的，而假设上周一个同事写的一个方法仍然一样有效，则不那么安全。

> **了解你正在做什么样的假设，对它们进行严格的检验。**
>
> *Know what assumptions you're making, and examine them critically.*

关键是要了解你正在做什么样的假设，以及何时对它们进行严格的检验。进行这个工作的一个特别好的时机就是当你止步不前的时候——这可能是因为其中的一个假设掩盖了真实的运行状况。

连续性是如何连续的

早在20世纪90年代初，我开发过一个性能要求高的、跨平台的应用程序。它已经成功地运行在几个不同的共享内存的多处理器架构上了，所以再要求我将其移植到当时的最新的DEC Alpha处理器上时，大家都认为那是很简单的事。要是这样就好了。

我花费了数周时间查询上千行的日志文件，但仍然不能对我所看到的行为拿出任何解释。好像是一个CPU以不同的顺序读取了另一个CPU发出的数据。但是，这不可能

是真的，是吧？

像几乎所有这个类型的其他机器一样，Alpha处理器实施连续缓存，保证每个CPU拥有一致的共享内存视图。而且我们已经假定"连续"意味着被一个CPU写入到内存中的数据将会被另一个CPU以它们最初生成的顺序看到。

在绝望中，我创建了一个非常小的（不到20行）测试程序，产生一对双线程，如果见到以不同顺序写入内存的东西，就会产生激烈反应。在几秒钟内它就产生了反应。这里的连续性并不是我们想象的那样——我们需要使用内存屏障，以保证非常重要的顺序正确性。[①]

3.4.3 多重原因

诊断阶段最常见的工作，就是你要寻找问题产生的原因，通常这是应该做的事情。正如奥卡姆剃刀告诉我们的，简单的解释往往是最富有成果的，假设是某个原因产生了问题要比假设是多个原因产生了问题简单得多。

不过，正如我们已经看到的，奥卡姆剃刀只是一个原则，有时事情确实是很复杂的。

> **有时，事情确实很复杂。**
> *Sometimes, things really are complicated.*

面临多种原因的最常见信号是一种你处于模糊状态的感觉——发生了一些似乎没有明显解释的奇怪的事情。

最富有成效的解决多原因缺陷的办法是对问题进行隔离，并找到一个方法来重现缺陷，重现的缺陷产生的原因只依赖于多个原因中的一个，而不依赖于其他原因。这样做的难易程度取决于你对问题做了多少诊断。如果你已经有把握问题会存在于哪个地方，那就可以帮助你构建一个替代性的缺陷重现方法，来回避另一个缺陷原因。

另一种方法是开始先找寻同一区域内其他较明显的缺陷。处理这些缺陷有时可以扫清障碍，让你理解得更透彻，使初始问题更加凸显。

如果这些方法都没有起到作用，那么深吸一口气——这回你算遇到了对手。你将不得不像以前那样继续你的诊断，同时还要记住你的实验可能无法预测，因为它们可能会被不止一个潜在的问题所影响。这时没有一个人会说调试很容易。

① 现如今，我们都已习惯于使用能重排序以提高性能的、高优化CPU，但在当时，这还是一个新兴事物。

3.4.4 流沙

“模糊”感觉产生的另一个原因是一个不断变化的基础系统。我们依赖的实证方法的基础是我们可以一次又一次地重现问题，并且每次获得相同的结果。没有了这种确定性，想取得进展是极为困难的。

这时，你一直记录在案的已尝试的手段和获得的结果（你一直在记录吗）一下子就可以派上用场了。如果你重新运行了一个实验，而今天得到的结果与昨天的不同，这就是个很好的迹象，表明在这段时间内肯定发生了什么变化。

> 面对一个不断变化的基础系统，停下手头工作并弄清楚是什么在变化，为什么变化。
>
> *If faced with a changing underlying system, stop and work out what's changing and why.*

如果你怀疑自己遇到了这个问题，请立即停下来，继续进行只会陷入更大的麻烦。你的主要目标就是准确地查明究竟是什么在变化，以便你能控制它。

最明显的就是那些与软件交互的诸如数据库或第三方系统的东西，但请记住，你的软件的运行可能被太多不同的因素影响。也许你现在没有足够可用的磁盘空间，没有足够的空间来容纳一个临时文件？或者你安装了一个新的软件程序包，它更新了一个系统库文件？或者，如果你的软件每天根据钟点安排运行情况，它甚至可能只是因为时间改变了而反应不同？

有12个月意味着什么

我还记得我的第一次团队合作经历，那是在读本科期间，我们必须完成小组编程任务。这件事对我的教育意义不止一条。

我的团队中有一名成员，坦率地说，没发挥任何作用。我们把他编写的所有代码都丢弃了（不是说他写多了），只留下了一样——一个返回每月多少天的函数。

总之，代码运行得很好，我们及时地提交了代码并通过了所有的测试。然后，我们的教授联系了我们，说他每次执行的时候都会崩溃。我们非常迅速地跟踪到那个我们没有改写的函数，是它导致了问题，如下所示：

```
int days_in_month(int month) {
  switch(month) {
    case APRIL: return 30;
    case MAY: return 31;
  }
}
```

我们5月提交的代码，但是我们的教授直到6月才开始对代码进行评价。所以，程序就崩溃了。

3.5 思维游戏

调试是很艰苦的，有时简直苦不堪言。在职业生涯中，你保证会遇到根本就看不到前进方向的情况，至少有一段时间会这样。

有时看上去好像软件的运行状况是根本不可能的，每一个证据和你看到的都互相冲突。如果不是亲眼所见，你可能会发誓说这是不可能的。

在另一些情况下，你所调查研究的各种方法都变得毫无价值，而你又想不到别的方法。

不要灰心。请放心，我们都遇到过这种情况——而且还会反复遇上。这只是软件开发工作的一部分。最终你会找到一种解决方法的。

如果你遇到障碍了，下面这些技巧可以帮助你解决这些问题。

3.5.1 旁观调试法

最有效的一个扫除障碍的策略就是向其他人求助。让一个已经几小时（或几天、几周）没有接触过这个问题的人去审视这个问题，可能会带来新的思路。即使他们不能立即发现问题，但是人多力量大，问题很有可能就会得到解决。

这是怎么回事

杰瑞米·塞迪克

我两岁的儿子艾丹有一次发现了一个缺陷。他指着屏幕上我已经找了一二十分钟的Lisp代码说，他发现屏幕上有的缩进模式和其他的地方不一样。

但是，任何做过这项工作的人都知道，往往把问题再解释一遍的简单行为也能激发灵感。有时帮助你的人甚至不用说一个字，他们就像张硬纸板呆在那里听你讲就够了（或者说像橡皮鸭、木头人，或任何其他无生命的对象）。



小乔爱问……

如何做到旁观者清

虽说旁观者清，但实际上如果帮助你的不是活生生的人，那么这个旁观者能起的作用就非常有限。有些人可能能够让他们的猫真的听懂他们所说的话，但是我们大部分人都对此持怀疑态度。

因此，要做到旁观者清，需要做到以下几点。

- 注意。如果你用心倾听，你所“帮助”的人会很清楚。
- 提问。解释不清之处是一个信号，表明其中很可能含有一些未经仔细考虑的假设。
- 留意没有探究过的地方。不要以为对你来说很明显的事物对你在帮助的人来说也是明显的。他们寻求你的帮助是因为他们遇到困难了，而且通常我们会在一些看似显然的地方感到迷惑。
- 尽力了解接下来将会发生什么。如果了解了，你可能会提出更好的问题。而是这种灵感可能启发了你解决问题的思路，而对方是当局者迷。



小乔爱问……

如果我找不到可以交谈的人怎么办

如果你找不到任何一个人来扮演旁观者，也不是就一筹莫展了。试着在纸上描述一下你的问题或者给朋友写一封电子邮件。不需要审查自己——就像一个作家写作那样。

解释问题会帮助你理清思路。

Explaining the problem helps get your thoughts in order.

要这么做的理由很充分。向其他人解释问题会强迫你去理清你的思路，列出你的假设，并构建一个从基本原则出发的观点。很多时候，只有设身处地才能找到解决方法。如果找不到，你也不会失去什么啊！

3.5.2 角色扮演

角色扮演在解释和探讨问题时十分有用，特别在涉及那些互相独立的系统之间相互作用的问题时。“你扮演客户端 1，我扮演客户端 2，弗雷德可以扮演服务器。现在我们如何建立客户端间

会话？”

如果合适的话，不要忘记使用一些工具。索引卡可以代表网络上信息的交换，或者凡持有充气“企鹅”娃娃的人都拥有数据库锁。我待过的大多数开发人员的房间都堆满了很多年来从不同的展会上收集的零碎物件，让它们替你改头换面。

3.5.3 换换脑筋

你花费了令人沮丧的一天，却在一个棘手的问题上没有取得任何进展。你非常郁闷地度过了一天。那天晚上，当你正在做一些与问题完全无关的事情时，解决方案突然浮现在了脑海当中。当你做饭的时候，在电话中和妈妈说话的时候，给孩子读睡前故事的时候，你的潜意识正在逐步地解决这个问题，并且它再次显示了自己的魔力。

我们所有人都经历过这种事情：突然间恍然大悟，以前模糊不清的突然变得完全清晰了。不好的是没有办法特意设计出这种效果。有时，你的潜意识会帮助你完成工作，而其他时间它将继续保持沉默。但是你一定可以做些事情来帮助你发挥潜意识。

> **让潜意识帮助你。**
> *Help your subconscious help you.*

如果你发现自己变得沮丧或超负荷了（进行了大量工作却几乎没有进展），这就表明你需要休息一下了。换一个问题做一段时间，喝杯茶，散散步，娱乐一会儿。任何能把你的思维从问题中转移出去的事情都可以。

最坏情况下，你头脑清醒地重新返回这个问题，但这回更有可能取得重大进展。最好的情况是，如果你幸运的话，奇迹将会发生，你的潜意识会帮助你完成工作。

如果潜意识不期而至，那么把它写下来。如果没有笔和纸，那么给你自己发一条短信，或者告诉你恰巧遇到的任何人——没有比第二天想不起来的事更令人沮丧的了。

如果遇到特别困难的问题，可以多休息一会儿。第二天早晨起床后，看问题的角度可能就大大不同了，没准会对解决问题起到不可估量的帮助作用。但要注意别矫枉过正——跟踪一个缺陷意味着你需要了解很多不同的东西。休息太长时间后，你可能需要重新进入角色。此外，还有一些缺陷是没有解决捷径的，只能一鼓作气战斗到底。

3.5.4 做些改变，什么改变都行

前面讨论过，认真仔细地思考你的实验是非常重要的。你应该知道为什么做这些实验，你想

通过它们得到什么样的结果。

但有时候，如果你完全陷入困境之中，做一点改变是十分必要的。任何改变都可以。也许它不会告诉你任何东西，但有时它会让你感到惊奇——让你惊奇的事总会教给你一些东西。

我想要的东西根本没有出现

马修·罗迪·雅各布

我最近遇到过一个缺陷，这个缺陷似乎是“一个表单的多选控件间歇性地不能自动选择”。它在空值与我想要选的值之间跳跃。

每次我重现问题的时候，都使用同样的输入值（Fire 和 Health）或保持输入域空白。

过了一段不知所措的时间后，我尝试输入了不同的数据集（Charities 和 Probation）。结果让我很吃惊，它仍然断断续续取 Fire 和 Health 中的一个值或者为空。这个问题和默认设置没有任何关系，是因为表单被缓存了（在 2 台不同的服务器上）。我原来没有找到这个原因，是因为我总是选择相同的选项。

3.5.5 福尔摩斯原则

当你排除了一切不可能后，无论剩下的是什么，无论它有多么不可思议，它也一定是真相。

When you have eliminated the impossible, whatever remains, however improbable, must be the truth.

福尔摩斯有一句名言：“当你排除了一切不可能后，无论剩下的是什么，无论它有多么不可思议，它也一定是真相。”

这是一种宝贵的提醒，虽然多数情况下最有可能得到的是简单解释，但是有时事情的真相真是不可思议。有时，太阳系所有的行星真的以正确的方式排列成行。不要只是因为一个解释感觉太不可能为真而拒绝它。

它永远是大管家

弗里德里克·张

我的 Ruby on Rails 应用程序中的一个控制器声明其 start()方法不存在。这段代码已经几个月没有改动了，我可以在源代码中看到方法的定义。那么，是什么妨碍了 Rails 找到这个方法呢？

我启用了我认为可靠的调试器，对 ActionController 进行了逐步调试。出于安全考虑，并非所有的方法都作为 action，因此 Rails 清除了所有定义在 Action-

Controller::Base 中的内容。由于某些原因，ActionController::Base 突然获得了一个名为 start()的方法——迷局解开了。

我找不到这个 start()方法来自何处。它肯定不在源代码中。我启动了交互式控制台来作一下深入的调查，更加奇怪了——没有 start()方法，即使我运行同样的代码也没有。

在经过了大量的调试并且无计可施时，突然间我想到问题是不是调试器本身产生的。我看了一下调试器的源代码，果然，它定义了 Kernel#start()方法，这个方法又引入到 ActionController 中。因此，造成了 action 的时有时无，看似随机的因素其实取决于我是否调试了其他的代码。

3.5.6 坚持

虽然有时看起来不是这样，但实际上任何一个缺陷都是可以被诊断的。运行在计算机上的任何软件都是人类创造的，我们可以随时提取足够的信息来准确地了解它的运行状况。从这个角度来说，软件几乎和人类从事的任何一个其他领域都有所不同。

当然，这绝不意味着诊断很简单。当你对自己究竟会不会弄清目前的问题感到绝望时，请记住，总有一个方法可以帮你解决问题。只要有足够的时间，付出足够的精力和决心，你一定会解决问题的。

3.6 验证诊断

我们人类是多才多艺的动物。不幸的是，我们的才能之一是自欺欺人——如果想要什么是真的，我们就非常善于使自己相信这些是真的。考虑到这一点，花时间验证你的诊断是否真正站得住脚是很值当的。

- 向其他人解释你的诊断。他们可能会发现一个缺陷，即使是旁观者效应也可以发挥其神奇的作用来帮助你达到这样的效果。
- 检查源代码的原始副本，不要带上你所做的任何修改，验证你编程时的分析仍然正确。你可能已经注意不要引入任何意外的不良结果，但是只有从一个已知的、没有问题的副本代码重新开始，才能让你对成功抱有信心。
- 既然你明白了这个问题，有没有其他方法可以证明它确实以你设想的方式进行工作？快试试。你看到了预期效果了吗？
- 多和他人讨论，并假设你是错的。知道你犯过什么错误了吗？

这些检查和平衡不会花费很长时间，我只是希望它们能说服你最终你是对的。如果没有，那么，它们可以使你免受尴尬并节省时间，确实非常值得一试。

既然有了一个你信任的诊断，所有剩下的工作就是进行修复，而这正是我们将在下一章讲述的内容。

3.7 付诸行动

- 构建假设，并用实验来测试它们。
  - 确保自己明白你的实验要说明什么。
  - 每次只做一个修改。
  - 把你尝试过的工作记录下来。
  - 不要忽略任何事物。
- 当事情并不顺利的时候，做到以下几点。
  - 如果你做的修改似乎没有产生效果，那么你就没有改到点子上。
  - 验证你的假设。
  - 你是否面临多个有内在联系的原因或一个不断变化的基本系统？
- 验证你的诊断。



第 4 章

修复缺陷

到这里，问题的诊断就完成了。现在可以庆贺一下了——很可能你已经完成了任务中最困难的部分。既然知道了问题的所在，修复它应该是一件轻而易举的事。

但是，还是要小心。到目前为止，你的工作重点一直是在努力弄清楚软件究竟是怎样运行的，以及它为什么会出现问题。你已经创建了特定的实验，修改了代码来插入日志功能，找到出错条件，或者通过其他方式使软件按照你的要求运行。在思考问题的时候，你已经形成了一个谨慎并具有创造性的、开放的思想方式，随后你已对不同的假设进行了论证和反向论证。

现在，你即将开始一个完全不同的实验。“一切都可以运行”式的诊断会被更加专业化的、结构化的方法所取代，这些方法需要对代码进行高质量的、精确的、可靠的修改。总之，你现在已经不是一个寻找缺陷的探测者了，是时候再次成为一名软件工程师了。

当然，你的主要目标是修复问题。但是对于一个好的修复来说，不仅仅是让软件正确运行——你还需要为将来奠定良好的基础。如果不小心，软件可能会取得相反的效果，也就是常说的“比特腐坏”（bit rot）的状态中。一次次修复，一系列零散的未经过仔细考虑的修改，都将使原本简洁的设计逐渐消失。

> 对于一个好的修复来说，不仅仅是让软件正确运行。
>
> There's more to a good fix than just making the software behave correctly.

在这一章中，我们将探讨如何同时实现以下目标：

- 修复问题；
- 避免引入回归；
- 维持或提高代码的整体质量（可读性、架构、测试覆盖率和性能等）。

4.1 清除障碍

在深入解决问题之前，有些基础工作是必须要做的。第一步就是要确保一切重新开始。

当出现问题时，你很可能匆忙地修改了源文件并且重新进行了配置，并创建了实验，把数据文件散落得到处都是。你肯定不想突然地签入[1]这些随机的修改。如果不在签入之前弄清楚问题，你会面临难以将它们区分开来的危险。

但是，你不想轻易地抛弃一切，因为你所做的一些修改或者你在诊断阶段创建的数据文件，很有可能会为测试用例的编写奠定良好的基础，而这些测试用例又是修复缺陷的一部分。此时，你的源代码控制系统照例很有用处。

首先，你需要对所做的修改进行快速审核。[2]不要跳过这一步。经常发现这些已经忘记的修改会让你惊讶不已。

> 从一个干净的代码树开始。
> *Start from a clean source tree.*

通常情况下，丢弃这些修改将是正确的选择。[3]然而，如果要留住这些修改，不要把它们留在原地并修改附近的代码。请记住，我们的目标之一是避免回归，这些修改不是以这样的方式存在意味着它们是可以信任的。随时做记录，或保留相关文件的副本，但是当你开始实施一个修复程序时，从一个干净的代码树开始是很关键的。如果在诊断阶段所作的修改很深入，你甚至可以更方便地将整个新源代码树进行更新，并把错误的代码树放在旁边作为参考。

这给了你一个值得信赖的起点。接下来要考虑的事情是，如何去证明你的修复确实解决了遇到的问题。

4.2 测试

假设你的开发过程包括测试优先（或测试驱动）开发，因此你拥有一个自动化的测试框架和大量的单元测试工具。现在，当你要对源代码作出更改时，这种方法确实能够收到良好的效果。你不仅可以用它来确保你的修复程序会解决此问题，而且它为避免卷入回归提供了很好的保障。

> 首先确保所有的测试都是通过的。
> *Start by ensuring that all your tests pass.*

因为你要依靠大量的测试，首先确保它们都通过（当然应该是这样，因为你刚刚确保了使用的是干净的源代码树）。如果它们没有全部通过，请立即停止，并找出原因。或许因为同事签入了一个有缺陷的更改？或者你的本地环境配置不正确？无论如何，如果测试没有通过，就不能帮助你进行即将作出的更改。

---

① 签入，用在版本控制中，是指当某个人修改完代码后，就签入到版本控制软件中，别的人才能获取最新版本。——译者注

② 如果你使用 Subversion，可以在 `svn diff` 命令后使用 `svn status` 命令。

③ 使用 `svn revert -recursive` 命令。

测试优先开发的规则之一是，直到有一个失败的测试时，才应该更改源代码。因此，你需要证明你已经拥有了通过所有测试的测试套件作为坚实的基础，同时最好确保有一个失败的案例。

如果缺陷被遗漏了，很显然，要么现有的测试程序未能充分地测试有问题的功能，要么测试程序本身就有问题。因此，要么需要增加一个或多个新的测试程序，要么修复已有的测试程序。

下面是应该采用的顺序。

(1) 运行现有的测试程序，并证明它们能通过。

(2) 添加一个或多个新的测试程序，或修复现有的测试程序，以显示错误（换句话说，就是失败）。

(3) 修复缺陷。

(4) 证明你的修复起了作用（那些失败的测试不再失败）。

(5) 证明你没有引入任何回归（以前通过的所有测试现在都没有失败）。

当然，现实中，根据缺陷修复的复杂程度，测试的过程不一定是按照上面的顺序线性进行的。你可能要在建立测试和修改代码间反复多次，才能确定最终解决方案。

你创建的用来重现和诊断问题的实验、数据文件以及其他任何东西形成了丰富的思想源泉。事实上，如果运气好，所有你需要做的就是整理并规范好你所创建的东西。但是请记住，我们要的是具有产品质量的东西。当你没有真正理解问题的时候，创建的测试也有可能是一个很好的出发点，但是你应该花些时间来确保它们得到了很好的构建，并且测试了一切应该测试的东西。

如果不采用测试优先的开发过程呢？即使这样，测试仍然是至关重要的。如果你没有一个可靠的测试来证明问题的存在，如何确保你已经解决问题了呢？主要的区别只是，你可能是用手动的而不是自动化的方式来进行测试，而且测试结束后就丢弃了。在缺少回归测试的情况下你必须非常谨慎，因为意外引入一个回归的机会要高许多。

> 在设计你的修复方案之前，确保你已经知道如何进行测试了。
>
> *Make sure you know how you're going to test it before designing your fix.*

测试套件如何挽救了我

多米尼克·宾克斯

我过去一直用 PHP 编写应用程序。幸好我们有一个用途广泛的自动化测试套件。我作了修改（一个非常简单的更改），结果，在一个我没有接触过的 Web 服务接口测试时出现问题了。我所作的修改肯定与它无关。

唉。

原来，测试失败的原因是Web服务返回无效的XML文档。我查看了XML文档，看起来没有什么问题。我不是XML大师，所以我找了一个同事去分析，他也说没有问题。我一直听人说XML是比较简单的，所以应该不难看出其是不是正确的。

最后我在XML文档的开头找到了一个换行符，这在XML里是违法的，因此报告XML文档无效。

怎么会出现换行符呢？嗯，我不小心将一个换行符添加到文件末尾的<php>结束标记后面。其结果是，PHP处理器把它当作HTML的一部分发送到“浏览器”。

一般而言，一个多余的换行符不会产生任何影响。然而，当PHP代码被添加到在Web服务代码路径中时，换行符在其余XML文档之前出现，这导致了XML无效。

如果没有自动测试，程序也许就会当作产品发布了，然后再不得不送回来以弄清楚为什么某部分服务会出现问题。

4.3 修复问题产生的原因，而非修复现象

几年前，我用C语言编写嵌入式代码时，跟踪到某个函数出现的缺陷。该函数类似于如下代码：

```
int process_items(item* item_array, int array_size)
{
    int i;

    /* 不知何故，传入的array_size会差1，这里给它加上 */
    array_size++;

    for(i = 0; i < array_size; i++) {
        «Process item_array[i]»
    }
}
```

这名开发人员（我就不提名字了）几个月前就已经确定是因为array_size错误赋值而产生了缺陷。然而，他没有继续分析确定调用该函数时传递错误参数值的原因，而是通过修改这个函数的内容来解决这个问题。

果然，我随后发现，process_items()方法可能在多个位置被调用，但偶尔array_size的值也是对的，这就会导致数组溢出（C语言）问题，而这个问题又导致稍后出现了一些令人费解的问题，需要花费相当大的精力去跟踪。


小乔爱问……

“掩盖真相”到底会不会有问题呢

有时即使我们明白问题的根源，还是有可能“掩盖真相”。这么做的原因有以下几点：(1) 这个缺陷是深深植根于结构的，彻底修复这个缺陷将涉及大面积的修改，而且很危险；(2) 会带来与以前版本是否兼容的风险（见 8.2 节）；(3) 做一次真正的修复比更新一个好的应用补丁需要付出更多的精力。

然而，这是一本重实效的书，因此，必须承认有时这不是正确的方法，不过这种情况确实非常罕见。

每一次决定不解决问题的根源的时候，的确是大大降低了代码库的整体质量。不仅如此，这还会产生不好的心理影响（见 7.1 节）。

是的，有些时候，选择不解决根本问题对我们来说可能更为合适，但是这只能作为最后没有办法的办法，而且还必须睁大双眼防止后患。

但这种事情会频频发生。我们所有人都将在职业生涯中的某些时候发现自己在跟踪这样的缺陷。

促使我们最终犯了这种错误的原因有两个。最常见的是因为我们的分析远远不够，还没有发现问题的真正根源。但有时是由于无法应对时间进度方面的压力所导致的。

首先关注一下时间进度方面的压力。在某个地方，一个软件工程项目可能在没有固定的时间进度压力的情况下运作着。不过迄今为止我还从来没有从事过这样的项目。即使你足够幸运，在这种情况下工作，受你缺陷影响的用户也不可能愿意为了你修复缺陷而多等一分钟。

其结果是，你很可能会迫于压力而只是“让缺陷消失”，然后去进行下一个任务。在平时可以信誓旦旦地说绝不做这样的事情，但在非常时期，一边是愤怒的用户在向你大嚷大叫，一边是不耐烦的项目经理给你脸色，你就很可能做这样的事。

无论这么做有多不好，比起在完全了解问题的根本原因之前就进行修复的风险，都是微不足道的。至少，如果你了解根本原因，采取一个明智的（即使被误导了的）决定并迅速地实施解决方案，你是了解你所做事情的意义的。如果你并未真正了解程序的运行状况，那么根本就不可能预测到你的行动会带来什么样的后果。

回想一下，解决当前这个问题只是我们制订的三个目标之一。我们还需要避免引入回归并保持代码的整体质量。我们的重点往往是第一个目标，但后两个同样重要（从长远来看，可能更重要）。考虑到这一点，做一些自己尚不理解的改变是十分不明智的。

你怎么知道你真正了解了问题的根源？好吧，要用到一些经验法则。（例如，给同事解释这个会理直气壮吗？解释时会用"由于某种原因……"或"我不知道为什么，但是……"这样的短语吗？）但是，这个问题最简单的事实是，大部分情况下你知道自己是否了解问题的根源。这就是所谓的学术诚实——勇于正确对待自己，即使你似乎找到了一种修复缺陷的方法，但如果你还不能够确信自己是否真正理解了问题的症结所在，那么你也不能相信你的修复。

4.4 重构

最近的几年里，敏捷开发方法的日益普及给软件开发带来了巨大的变化。对于代码结构（而非项目管理）而言，最显著的影响就是两种技术的广泛应用——自动化测试和重构技术。

重构是改善既有代码的设计而不改变其行为的过程。后者有时会被忽视，但却是这里我们主要关心的方面。对于重构的完整介绍，请参考 Martin Fowler 的经典著作《重构：改善既有代码的设计》[①]。

修复缺陷往往不会涉及重构。你正在使用的代码中包含缺陷，这一事实本身往往意味着代码本应得到更简洁或更好的构建。你很有可能会找到哪些代码能够改进。

而且，如果缺乏经验的话，在为修复缺陷而作出必要的修改时会产生重复，而这些重复是不应该有的。（这就是《程序员修炼之道》[②]一书中描述的"不要重复你自己"（Don't Repeat Yourself, DRY）的原则。

执行这些重构和修复缺陷同样重要（请记住，我们的目标之一是维持或提高代码的整体质量）。有时候在修复缺陷之后重构比先重构再修复更合理（因为这样做可以使你更容易修复缺陷）。有时，当从事一项特别复杂的修复工作时，你会反复地经历重构和缺陷修复。

> 重构或修改功能——要么选择前者，要么选择后者，绝不能同时进行。
>
> *Refactor or change functionality—one or the other, never both.*

但是记住，重构的同时绝不能改动代码功能，同样也不能修复有缺陷的代码。

由此引出修复缺陷时进行源代码控制的话题。

① 该书中英文版均已由人民邮电出版社出版。——编者注

② 该书中文版由电子工业出版社出版，英文注释版由人民邮电出版社出版。——编者注

> 小乔爱问……
> **重构的关键是什么**
>
> 很多第一次接触重构的人对重构的反应是"那又怎样"。这只不过是代码的"整理"，这些代码已被先期的程序员做得差不多了。当然，在 Martin Fowler 写作《重构》一书时，某种程度上他所做的一切就是将已被开发者使用了多年的技术进行重新编目。
>
> 但是重构不仅仅是一项有用的技术编目，其中还包含 Fowler 的两个关键见解。
>
> - 只有在一个全面的单元测试套件的安全网中才可以安全地修改既有代码。
> - 我们不应该在试图重构代码的同时改变其行为。
>
> 换句话说，你可以修改代码的行为，也可以重构它，但不能同时做这两件事。
>
> 经过反思，很容易理解为什么是这样。假设你试图在重构代码的同时改变其行为，这样做后，有些测试可能会失败。这可能表明你在修改结构时犯了一个错误。或者，它可能是功能修改的预期结果。然而，很难确定是哪一个。对功能或结构的修改越复杂，就越难以确定失败原因。
>
> 只修改代码或只重构，就可以避免测试失败，从而充满信心地对大改动的代码开展重构。

4.5 签入

源代码控制系统是我们的武器库中最有力的武器。但是如果运用不当，会失去它的很多价值。

我们很容易将很多错误集合起来，将它们一站式签入。毕竟这样做可以一鼓作气解决所有问题，打破这个过程就太可惜了。但这样做会大大降低源代码控制工具的作用。

从调试的角度来看，源代码控制的主要价值是审核记录。如果有人确实引入一个回归，你可以通过对以前的修改进行查询（见 3.2 节）来准确地找到哪个修改导致了回归（因此是你需要修正的）。但是，这种方法的有效性与每次签入的规格大小成反比。如果存在问题的签入包含了对散落在整个项目的数百个文件的更改，那么即便发现是这一次签入导致了缺陷也是没有用的。

> **一次逻辑修改只做一次签入。**
> *One logical change, one check-in.*

为了避免出现这个问题，请坚持如下原则：一次逻辑修改只做一次签入。

对于简单的修复，这意味着需要一次单独的签入，但

大多数情况下会涉及多个逻辑修改（因此需要多次签入）。如果修复程序需要2个逻辑上独立的功能修改和2个独立的重构，这可能意味着4次独立的签入。

与以往一样，你应该自己判断。即使3次签入彼此独立，对需要3个单线更改的修复使用3次签入也可能矫枉过正了。但是如果能早些并且经常性地签入，你就会少走些歪路。请记住，确保你的签入意见尽可能具体且有意义，会让人受益匪浅。

> **在签入之前进行比较。**
> **Diff before check-in.**

最后一点，无论修正缺陷或实现新的功能，每次签入之前一定要弄清楚你要签入的是什么，这是很好的习惯。这个过程不会花太长时间，并且偶尔还会有意外收获：可能发现计划外的变化。

4.6 审查代码

不管多么小心，你都有可能制造出一个让事情变糟而不是变好的修复。尤其是当它涉及代码质量和可维护性的时候，即使进行再多的测试（自动化或以其他方式）也无法保证不犯错。正式或非正式的代码审查是个好办法，可以在问题产生永久性的损害之前就发现它。

代码审查是一些开发方法中固有的一部分。例如，通过结对编程，XP（极限编程）可以保证每个更改都有2个人进行检查。不过，不必把代码审查看作一个很正式的调试方法。

审查人和审查时间

代码审查没有固定的执行时间。有时可以让同事参与到修复的初期阶段，有时可能只是让他们在一个完成的更改上签字确认。

经验告诉我们，无论什么时候遇到具有不确定性或有风险的区域，都要做审查。请记住，审查不是一锤子买卖——只要合情合理，没有规定说你不可以反复地求助。

无论让谁来进行审查，你都可能受益匪浅。仅是让另一个人来检查你的工作就已经向前迈进了一大步。如果你在一个相对陌生的领域工作，去向更熟悉这个领域的人（比如代码作者）讨教会更好一些。相反，如果你很了解程序代码，那么找一个对此不熟悉的人来获得一个新的视角吧。

成功地修复缺陷是一个伟大的里程碑，但它不意味着结束。着手下一个任务之前，花点儿时间思考第一次缺陷是如何出现在软件里的。别的地方的其他实例是否有相同的问题呢？它可能再次发生吗？

4.7 付诸行动

- 缺陷修复包括以下三个目标。
  - 修复问题。
  - 避免引入回归。
  - 维持或提高代码的整体质量（可读性、架构、测试覆盖率等）。
- 从一个干净的源代码树开始。
- 确保通过测试后再做修改。
- 明确在做出更改之前如何测试修复。
- 针对缺陷的原因而不是现象进行修复。
- 要做重构，但永远不要与功能修改同时进行。
- 一次逻辑修改只做一次签入。



第 5 章 反思

缺陷修复天生是目标极其明确的。我们在处理的是非常具体的问题，很有可能的一种情况是，修复过程涉及一个孤立出来的代码区。尽管关注的范围如此狭窄，但是你还是需要有大局观。为此，你修复完缺陷之后非常有必要花些时间去反思一下。

在这一章中，我们将考虑以下几点。

- 这到底是怎么搞的？
- 这些问题是何时以及为何被遗漏的？
- 确保问题不再发生。

5.1 这到底是怎么搞的

每隔一段时间我的收件箱中就会收到一封标题很幽默的电子邮件，名为“缺陷调试的六个阶段”，内容如下。

(1) 缺陷不可能发生。
(2) 缺陷没有发生在我的机器上。
(3) 缺陷不应该发生。
(4) 为什么会发生？
(5) 噢，我知道了。
(6) 这到底是怎么搞的？

如同大多数的幽默一样，我们觉得滑稽是因为它反映了现实情况。特别是，你经常会在完成诊断之后还在想：“这到底是怎么搞的？”

如果你发现自己在这么想，就请稍微暂停一下。这在很大程度上表明你还没有真正完全了解缺陷所揭示的东西。继续思考下去，弄明白它究竟是怎么搞的，你极可能会在这个过程中学习到

很多东西。

并不是想象中的那么安全

> 我们有一套 Web 应用程序，这些程序将安全保护措施委托给共享的“看守人”程序，人们只用一个用户名和密码就能登录所有的程序。如果用户已登录到这套应用程序中的任何一个，他们不需要再次登录就可以使用其他应用程序，所有这些都由用户浏览器中存储的加密 Cookie 进行控制。
>
> 我在处理某个用户无法登录的缺陷——看来在某些情况下，生成 Cookie 的代码出错了。一旦我找出哪里出错，修复就是很容易的事了，于是又解决了一个缺陷。
>
> 但我有一种挥之不去的疑问。造成错误 Cookie 的环境十分普通，可为什么只有一个用户有问题？其他人如何就能够登录成功了？一定还有问题。
>
> 果然，深入调查表明，该系统并非我们想象的那么安全。它应该是更换密码定期加密 Cookie，但这并没有完全做到。那些遇到这个缺陷的用户并没有这样做，因为他们还能继续使用旧的 Cookie。
>
> 如果我没有听到内心中有个小小的声音说“还有点儿不对劲，你还没搞清楚”，那么我们就永远不会发现这一点。

肯特·贝克在《测试驱动开发》[Bec02]中谈到了类似的效果。有时，我们写一个期待失败的测试，但实际上它却通过了。当发生这种情况时，肯定有些东西是值得我们深入思考的。

但是这六个步骤中肯定也遗漏了一些东西。应该有第七个步骤：“它永远不会再发生了！”在接下来的章节中，我们将看看你能做些什么以确保它永远不会再发生了。

5.2 哪里出了问题

从缺陷中吸取教训的第一步就是要确定到底是哪里出了问题。

五个为什么

对缺陷发生的根本原因进行分析的一个有效手段就是问五次“为什么”，比如：

- 软件崩溃了，为什么？
- 该代码不处理数据传输过程中的网络故障，为什么？
- 没有专门检测网络故障的单元测试，为什么？

- 最初的开发人员并没意识到应该创建一个这样的测试，为什么？
- 我们的单元测试并没有考虑到网络故障，为什么？
- 我们在原先设计中没有考虑网络故障。

为什么是五次"为什么"呢？这只是一个经验法则——有时你需要的步骤很少，有时则很多。有时这也无济于事（它仅帮你识别已经知道的问题症结）。但总的来说它很有帮助，因为"五次"似乎在大多数情况下都适用。

5.2.1 我们已经做到了吗

是不是确定了哪里出了问题就是诊断要做的所有工作？是的，但我们这里所探讨的涉及面更广——软件最开始是如何产生这个错误的呢？

例如，你做出的诊断是，没想到接收服务器的数据时会产生网络中断，因而造成了这个缺陷。这是需要你诊断的时候要尽可能考虑清楚的问题。我们现在是要弄清楚为什么代码的开发人员一开始没有意识到他们必须处理网络故障。

5.2.2 根本原因分析

如果起初开发的代码中隐匿着缺陷，说明在你的开发过程中一定出现了什么问题。到底是在什么时候呢？又是为什么呢？

责备

前瞻性地找出工作过程中出现的问题对总体质量的提高是非常有帮助的。要注意，目标是汲取经验教训，不要责备任何人。

是的，某人在某些地方可能搞砸了，但我们都会偶尔犯一些错误。责备别人没有用。

责备文化具有腐蚀性，它会破坏成功最关键的因素——团队精神。如果人们担心会因为犯错误而被嘲笑或受到惩罚，那么，同事就会担心如何保护自己，而不去关心怎样尽力地为了他的团队或组织工作。在最坏的情况下，这甚至会导致说谎，涉及无辜的人，并导致其他不正常的行为。

榜样的力量是无穷的，无论是好榜样还是坏榜样。如果当你跟踪到一个特别棘手的问题后，就向主要责任人大声咆哮，那么团队的其他成员也会效仿。相反地，如果你自己造成的问题出现了，坦白并主动承认自己的过失，会告诉大家这没有什么好羞耻的。当问题出现时，你的处理方式比这个问题本身更加重要。

需求 这些需求是否完整且正确？它是模糊不清的、未被正确理解或遭到误解了吗？

架构或设计 有没有在架构或设计中存在什么疏漏——可能是我们没有考虑到一些事情吧？或者设计没有问题，但是我们没有正确地按照设计来做？

测试 我们对代码的测试是否达到了足够的覆盖率？或者可能测试本身就有缺陷呢？

构造 当你解决一个缺陷时这是最常想到的。或许开发人员在写代码时犯了一个很简单的错误，或许他们误解了某些基础技术（库文件、编译器等）。

5.3 它不会再发生了

一旦你确定了错误的来源，就可以采取措施避免它再发生。在某些情况下，这只不过是告诫自己将来在这一方面要更加小心，或者委婉地让你的同事认识到自己的错误；在有些情况下，你在事后反思时要加以总结——尤其是你发现了错误的模式发生在特定的点或特定的原因下；极少数情况下，可能需要给自己"敲响警钟"。

5.3.1 自动验证

我们应该关注那些出现问题的区域、常见的错误和同一问题的其他例子。假设你刚刚修复了会产生内存泄露的C++代码，如下：

```
void f(void)
{
    T* pt = new T;

    «Do something with pt»

    delete pt;
}
```

除非它调用的一个函数抛出异常，否则这组代码是不错的。在这种情况下，pt 不会被删除。这个问题有多种方法来解决，例如通过使用标准库中的 auto_ptr()方法，如下：

```
void f(void)
{
    auto_ptr<T> pt(new T);

    «Do something with pt»

    // auto_ptr ensures that pt is deleted even if an exception is thrown
}
```

很好，另一个缺陷油然而生。但在我们继续之前，应该考虑原来的缺陷是否是一次性的。至

少看起来很有可能原来代码的作者不知道如何编写异常安全的C++代码。在这种情况下，是否会有相同问题在其他地方出现呢？现在是该做审查的时候了，看看是否还有与该问题类似的其他例子，发现并解决它们，而不是等那些可能潜伏的缺陷被发现时才采取措施。

和同事交流

让同事知道他们犯了错误可能是一件很危险的事。一方面，这是非常有价值的信息，你有责任让他们知道，使他们能在未来避免重蹈覆辙。另一方面，我们程序员并不善于人际沟通，如果你冒冒失失口无遮拦地说某个人把事情搞砸了，会很容易出现问题。

没有什么固定的技巧捷径。有时候，不管你是多么小心，你的善意得到的可能却是不友好的回应。但你还是可以主动做些有益的事情。

- 最重要的是，要有正确的出发点。如果你告诉他们犯了错误，只是为了自己得到优越感，那还是免开尊口吧。无论那些反馈信息说起来显得多么有帮助，你的真实动机迟早会昭然天下的。
- 在进行交流之前要三思。设身处地地想象一下面对同样情形时自己会是什么反应，并且还要清楚地知道每个人的性格都不一样。
- 避免妄作评价。可以说"我/我们怎样怎样"，而不说"你怎么怎么"。
- 要有建设性。
- 记住，你也可能说错了。不要简单地宣布，他们犯了一个错误，要尽可能地与他们探讨。你可能会发现，他们的行为有十足的理由，也许那不是他们的错，甚至是你自己误判了问题。

更好的情况是，你能找到一种方法来自动检测这种类型的错误，使我们避免以后发生类似的问题吗？在10.3节中，我们将讨论一种自动检测这类问题的技术。而事实证明，我们可以针对其他各种各样的错误都这么做。

不论大小，多数项目的特有问题会越积越多。如"创建新客户前要确保先更新账户表"之类的问题。只要有这类问题存在，就会有人出错。而且，除非你碰巧知道有这个问题。否则就不知该如何回避。第10章会介绍如何创建不知情情况下能够自动给出提示的自调试软件。

5.3.2 重构

另外要考虑的是，代码是否会使人误入歧途。如果你发现某一特定问题的几个例子，是不是因为结构或接口方面的原因使人们很容易反复犯同样的错误呢？

例如，你注意到人们往往会传递错误的参数到下面的C函数中：

```
void drawRectangle(int x, int y, int width, int height,
    bool border, bool fill, bool client_coordinates);
```

如果你想一下典型的调用可能是什么样子的，就能明白为什么人们在为正确的参数苦苦挣扎。例如：

```
drawRectangle(10, 10, 30, 50, true, true, false);
```

这很难自圆其说。但改变以下一些代码的定义：

```
const int NO_BORDER = 0x00;
const int DRAW_BORDER = 0x01;
const int NO_FILL = 0x00;
const int FILL_BODY = 0x02;
const int GLOBAL_COORDINATES = 0x00;
const int CLIENT_COORDINATES = 0x04;

void drawRectangle(int x, int y, int width, int height,
    unsigned int options);
```

这又意味着现在可以采用如下方式进行调用。

```
drawRectangle(10, 10, 30, 50,
    DRAW_BORDER | FILL_BODY | GLOBAL_COORDINATES);
```

这样代码就清晰多了（更不容易出错）。[①]

5.3.3 过程

我们刚刚看到的这些技术的优点就是它们十分明确。改进后的接口排除了错误用法，能有效避免再次犯同样的错误。自动化的检查总是会及时发现问题所在。所以，如果你能使用这种方法从根本上解决问题，那么当然应该这样做。

不幸的是，并非总是能够找到一种方法来完全消除错误，检查你的工作过程可能是你唯一的选择。

也许你需要看看你的需求文档的质量是不是过关？或者考虑引入设计审查？也许在代码审查时对照常见陷阱的检查列表会有作用？

5.4 关闭循环

你正在着手的项目会有自己的一套规范：

① 如果你足够幸运，使用支持命名参数的语言在工作，那就不必这么绕圈子。

- ❑ 编码规范
- ❑ 测试规范
- ❑ 文档规范
- ❑ 报告/跟踪过程
- ❑ 设计指南
- ❑ 性能需求

无论你何时修复一个缺陷，你都需要牢牢记住它们。你是否需要把更新最终用户文档作为修复的结果？或为下一个版本更改日志？工作是否需要针对特定客户或项目进行跟踪？你需要添加一条记录在你的错误跟踪包中吗？或者把它交给QA部门（他们需要哪些支持材料）？

好，就介绍这么多了。我们已经谈了缺陷的来龙去脉：复制、诊断、修复、反思。下一部分将从大局看调试，介绍如何发现代码存在问题，以及如何把缺陷修复与软件开发相结合。

5.5 付诸行动

- ❑ 花时间来进行根本原因分析。
  - ■ 在过程的哪个点上产生了错误？
  - ■ 出什么错了？
- ❑ 确保同样的问题不会再发生。
  - ■ 自动检查是否存在问题。
  - ■ 重构代码以避免不当使用。
  - ■ 和同事交谈，适当修改进程。
- ❑ 多方反馈，与其他利益相关者形成闭环。



Part 2 第二部分

从大局看调试

本部分内容

第 6 章

发现代码存在问题

在本书的第一部分，我们一开始就假定已经知道软件存在缺陷了。在这一章中，我们要看看在此之前还要做什么。

缺陷可以随时出现在软件开发生命周期中的任何一个阶段——从代码编写完毕到代码发布后的经年累月。理想情况下，你会自己尽早发现这些缺陷——越早发现就越容易修复，而且避免以后让用户揪住把柄。

然而，很多情况下，尽管我们已经尽了最大努力，但是客户还是遭受缺陷的侵扰。在这一章中，我们会讨论遇到问题后将要做些什么。具体来说，我们将介绍以下内容：

- 缺陷追踪；
- 与用户合作；
- 与客户支持和 QA 部门合作。

6.1 追踪缺陷

无论你在开发什么样的软件，都需要创建一些流程，通过这些流程用户可以告诉你软件出现了哪些问题（最终，通过这些流程你可以告诉用户如何修复）。

6.1.1 缺陷追踪系统

缺陷追踪系统的规模、范围和使用方法各有千秋。有些是简单的单用途系统，有些则是功能全面的工作流程管理系统——用来控制和记录软件开发过程的方方面面（缺陷追踪只是其中的一小部分）。然而，这些缺陷追踪系统的基本目标都是相同的。

- 首要的一点，它能确保我们不会遗漏缺陷。
- 通过提供一个缺陷报告的标准格式，可以大大增加把所有相关信息都包含其中的机会。

- 作为审查线索，缺陷追踪系统可以确保我们知道每一个版本中哪些缺陷还未解决，哪些缺陷已经被修复了，被谁、如何修复的。它可以作为软件发布的重要信息来源（我们甚至可以自动生成它们）。
- 让我们设定缺陷的优先级，并确定先解决哪个缺陷。
- 通过为各利益相关方提供一种互相沟通的方法，它可以确保每个人都能了解缺陷的当前状态，还能确保职责在团队之间传递时准确提供了所有相关信息。
- 作为一种管理工具，它提供了该项目的当前状态概况。
- 极少数情况下，我们也会选择不修复某个缺陷，在作出决策后我们可以把这样做的原因记录在缺陷追踪系统里，这样将来就不必再重复这一过程。

无论你的缺陷追踪系统多么好，它的作用还是取决于它提供的信息。

6.1.2 怎样才能写出一份出色的缺陷报告

我们都有这样令人沮丧的经历：不得不面对一个不能提供任何帮助的缺陷报告。报告中除了"软件出现问题了"之外，就没有任何有用的信息，也没有告诉你下一步该怎么做。所以，我们知道我们不需要什么样的缺陷报告。然而，在理想情况下我们应该从缺陷报告中得到什么样的信息呢？

乍一看，这是显而易见的——所有可以帮助我们诊断问题的信息都是必要的。遗憾的是，在我们未完成诊断这个过程之前，我们无法知道哪些信息是相关的，哪些信息是无关的。因此，一份好的缺陷报告包含的信息宁多毋少。



小乔爱问……

我需要采用电子方式跟踪缺陷吗

因为我们每天都要与缺陷打交道，自然就认为所有的问题都应该使用技术手段来解决。但是有时候，这么做却会很碍事。

如果你在一个小的团队中工作，没有太多的缺陷要跟踪，不需要提供远程访问你的缺陷数据库的方法，那么一种非技术解决方案（把索引卡贴在一个白板上？）很可能更适合你。

但是，不要把那些技术含量低的系统不当回事。负责任地处理缺陷是专业软件开发很重要的一部分，不要因为缺陷报告是手写的就不认真对待。

> 报告应该是具体的、明确的和详细的。
> A report should be specific, unambiguous, and detailed...

它应该是具体的、明确的和详细的。如果要显示错误信息，应该如何准确地描述呢？如果数据被破坏了，该如何呢？如何准确地找出导致问题发生的行为呢？如果输出是不正确的，那么是以什么样方式输出的呢？如果有相应的支持资源（如导致问题重现的输入文件，不正确的输出画面，等等），这些都应该记录在缺陷报告中。

> 报告也应该是最小化的、唯一的。
> ...but also minimal and unique.

与之前说的宁多毋少相对应，缺陷报告的内容同时也应最小化。如果要使用一个有 10 000 行的输入文件来重现问题，那么这个文件到底可不可以被削减？导致缺陷发生的一系列行为中哪些是关键的，哪些是可以忽略的？如果在软件一个版本中出现了缺陷，那么是否有还没有出现问题的其他版本呢？

与此相关的，缺陷报告也应该是独特的。如果问题已经报告了，再次报告是不太可能有帮助的（尽管可能会有额外的信息添加到现有的报告中）。

我最喜欢的缺陷报告

我开发的产品有一个能够捕获所有异常的处理器，当事情无法挽回时，它会显示“屏幕崩溃”。谢天谢地，这种事不是经常发生，但是当我们追查原因时它可以极大地帮助我们。

之后我们收到一份缺陷报告，上面写着：“崩溃屏幕没有取消功能。”

你必须把报告交给提交报告的用户，如果我们能够实现它，这个“取消”无疑将是一个伟大的功能！

6.1.3 环境和配置报告

几乎每一个缺陷跟踪系统都有其环境域。如果你在桌面软件上工作，缺陷跟踪系统可能会被用来记录出现缺陷的操作系统，对于 Web 应用来说记录的就是浏览器。

听着不错。这就够了吗？

有两方面的原因说明这是不够的。首先，大部分非技术用户通常不知道他们的应用环境。你妈妈知道她用哪种浏览器吗？你销售部门的同事知道他们已经安装了哪些 Windows 服务包吗？

其次，也是更为重要的，计算环境正在变得越来越复杂，且无时无刻不在互相关联。了解用户使用 Firefox 浏览器访问你的网站就够了吗？几乎可以肯定这是不够的，你可能还需要确切知道他使用的 Firefox 是哪个版本的，浏览器在什么平台上使用，已经安装了哪些插件，他们是否启用了 Cookie 和 JavaScript 等。

你可以通过为软件增加一个选项来记录任何一个影响其行为的环境因素来解决这个难题。当然，对于许多缺陷而言，大多数的甚至全部的信息都是无关紧要的，但是如果自动记录，我们就可以不费力气得到这些信息，也可以相信其准确性。当这些信息有用的时候，它们就是无价之宝。

自动收集环境和配置信息。
Collect environment and configuration information automatically.

你能再说一遍吗

感觉是在赎罪（有时感觉像炼狱），在我职业生涯的很长一段时间里为移动电话开发软件。如果你认为用户不知道笔记本上运行着什么，那你应该问问他们手机上都装了什么。

冷不防地问一下（不要看），你的手机是什么牌子？什么机型？你不能确定吗？现在就看一看吧，你可能仍然弄不清楚。在你搞不清状况的时候你该怎样找到答案呢？

现在，想象一下当客户甚至回答不出他们使用的硬件的最基本问题时，你该如何做技术支持工作呢？“嗯——它是一个银黑色的按钮。您听明白了吗？”

同样的论点适用于你的软件支持的任何配置选项。如果你提供了一种可以自动记录的方法，类似于“你确定你开启了功能 X”这样的问题就不会再出现了。

这类报告的一个很好例子是：被很多 Web 浏览器支持的各种不同的 about:URLs 配置。试着对 Firefox 配置 `about:config`（图 6-1），`about:buildconfig` 或者 `about:cache`，你就知道我的意思了。

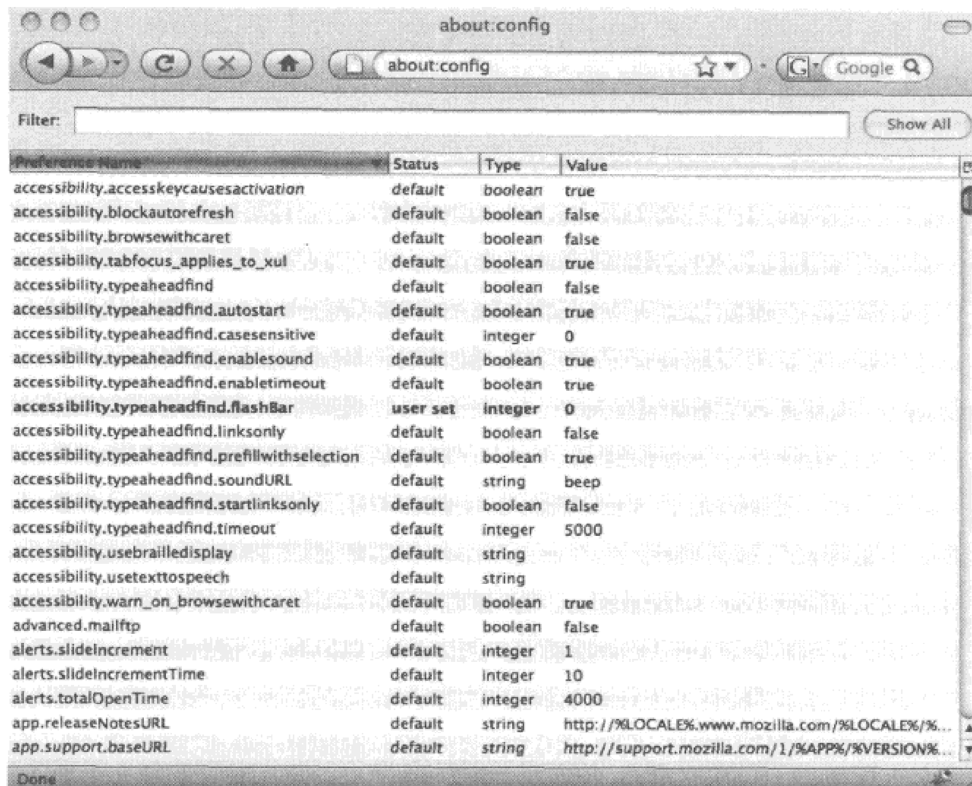


图 6-1 Firefox 的 ABOUT:CONFIG 页

6.2 与用户合作

作为一个软件工程师，你了解缺陷报告的价值。如果没有人花时间和精力向你反映问题，你就不能找出这些缺陷，也就不会修复这些你所不知道的缺陷。

6.2.1 简化流程

不幸的是，没有什么可以保证用户能花时间来报告缺陷，即使报告了也不能保证他们高质量地报告这些缺陷。但是你可以尽可能多地消除不利条件来提高缺陷报告的质量。

明确说明如何报告一个缺陷 在软件的“About”对话框中、在线帮助中及网站上和其他任何你认为合适的地方写上一些说明（或者最好直接使用链接）来解释如何报告缺陷。

自动化 安装一个顶层的异常处理程序，给用户发送缺陷报告的选项，报告中可以自动包含所有相关细节。

提供多种选择 有些人可能喜欢使用电子化的方式报告缺陷，而其他人可能喜欢通过与人交谈来说明缺陷。有些人可能喜欢电子邮件的方式，而其他人可能喜欢在线表单的方式。

要尽量简单 你要求用户执行的动作每多一次，能够完成任务的人数就会减少一半。换句话说，要求他们点击3次的话，就只有12.5%的用户能够完成。点击五次的话，这个数字会减少到3%多一点。

模板不要太死板 为缺陷报告做一个标准模板是个好主意。但要注意不要使该模板太死板，请确保你为每个域设置合理的选项，包括“以上都不是”。

尊重用户的隐私 你的用户的数据属于他们自己，而不属于你。你要清楚地了解这一点，并遵守相关的隐私策略。

小乔爱问……

我可以相信用户告诉我的缺陷吗

你会这样想，不是吗？毕竟，他们在使用这个软件，因为他们要用它完成一些事情，而缺陷妨碍了他们。

但是，不管你怎么想，大多数用户在软件出错时是不会来找你的。有些人会认为这是他们自己的错——他们“点错了按钮”。有些人会无可奈何地叹息（低声地诅咒），重新启动软件，继续自己的工作。还有一些人会十分费劲地试图找到一个缺陷的解决办法，而你可能在几秒钟内就能搞定。

作为一个经验法则，对于每一个向你报告问题的用户，将对应有10至100个用户经历过同样的问题，却没想到联系你。

6.2.2 有效的沟通

与客户交流可能会非常棘手。有效的沟通需要双方共享信息，但你的观点一定是不同于你的客户的。他们不能像你一样深刻理解程序代码，而你可能不会理解他们所从事领域的问题。你们使用不同的词汇，具有不同的技能，并运用不同的办法解决问题。你必须意识到这些差异以及这些差异可能引发的问题。

没有什么简单的办法能够解决这些沟通方面的问题。所有你能做的就是要明白这些问题是不可避免的，并且保持冷静，去解决这些问题。

1. 心智模式

我们通过创建特定的心智模式来应对这个世界。作为软件工程师，我们尤其知道这一点——软件是这些模式的具体化。

> **从用户的角度设想一下会发生什么。**
>
> *Imagine how things might appear from your user's perspective.*

你的客户也拥有他们自己的心智模式。你可能会惊讶地发现，他们的心智模式与你的有多么地不同。

当你认为你们是基于同一个模型而工作时就会产生风险。这会导致无数的误解，需要大量工作才能消除。

最有效的解决方法是换位思考，从用户的角度设想一下会发生什么。你的目标是将他们的观察结果（你可以信任的）与他们的解读（这些解读会带上他们的心智模式的色彩）分开。

我知道问题所在了

马库斯·格罗伯尔

我用语音合成技术为盲人开发手机软件，有时听他们说："我的电话在做某事的时候挂断了。"但我知道他要表达的真正意思是："在我做某事的时候音频输出中断了。"对盲人用户来说挂断与听不到声音是没有区别的。

2. 和非技术人员沟通

除非你在极个别的情况下为其他的程序员开发软件（这时你或许会遇到稍稍不同的沟通问题，但同样具有挑战性），否则你的用户可能并不了解所用的技术。他们可能不明白你认为理所当然的事情，也体会不到你在诊断一个问题时所涉及的微妙的细节。

最大的问题通常是抽取正确的细节问题。最不起眼的细节可能就是最最关键的地方，但是你的用户可能不会意识到这有多关键。他们极有可能会引用错误的信息，而不是准确地描述缺陷或者剔除一些"无关"的细节。当你更深入挖掘一些细节时，他们可能也无法很好地响应你。

唯一的解决办法是要有耐心，解释一下为什么这些细节很重要，并通过必要的步骤说服他们来收集所需要的有关数据。这可能是令人沮丧的，你在几秒钟内能解决的事情却可能需要他们很长的时间，但是你付出的时间绝对是值得的。

你交谈的对象有多专业

万迪·马塞

通过电子邮件或电话判断一个人有多精通于技术是特别难的。当我们和用户交谈时

这是一个要处理的问题。如果他们精通计算机操作，而你告诉他们说“找到蓝色的图标E”来启动IE浏览器，就会使他们非常恼火，但是同样令人吃惊的是，我们最后还是不得不总用这类语言与用户交谈。

我曾经对一个开发EPOS系统的公司非常地不满意，那次我报告了一个与他们的报告有关的问题。在他们的“每日最终”报告中有一个数字总是有错误。他们只是告诉我，他们知道自己在做什么，而我是不正确的。他们笃定我是一个用户，于是也“不专业”，故而也不可能得出正确的判断。甚至当我告诉他们我是培训会计师（因此知道得很清楚，“每日最终”财务报告上的这个数字是错误的），而且对软件开发也有一定程度的了解的时候，他们仍旧根本不把我说的话放在心上。这个问题一直没有得到解决，因此我们只有采用其他变通方法，因为我们能够这样做。然而，那家公司却给我留下了傲慢、愚昧、不信任用户的持久印象。我再也没有费事与他们联系提出任何疑问，更不用说升级建议。而且我们再也不会购买他们的任何软件。

误解不是用户的专利

具有智慧是成为一个好的软件工程师的必要条件。我们中的大多数人在学校里都有很好的成绩，都为自己的智慧而骄傲，但这并不能保证我们不会犯错。

当存在非常明显的误解的时候，请记住，有可能是你出错了。你可能对软件更了解，但在应用领域你可能并不会比用户理解更好，因为这是他们的专长。

3. 发布你的缺陷数据库

让你的缺陷跟踪系统对于所有的用户来说都是可用的。[①]如果之前你并没有做这方面的事情，那么让大家看到你的“爆料”可能是很可怕的事情，但它带来的好处是巨大的。

- 鉴于你认真对待用户的报告，及时给予反馈，最终解决报告中的问题，这样会给用户信心，让他们觉得确实值得花时间来做好报告。
- 如果用户在报告缺陷之前就能搜索你的缺陷数据库，就会大大减少重复的缺陷报告。
- 一个用户看到别人的缺陷报告时可能会深有体会，这便会提供重要线索让你解开他的难题。
- 了解已有的缺陷报告对那些并不知道如何编写缺陷报告的用户来说是很好的办法。

① 如果你使用托管的方法的话这是非常容易实现的（A.1节提供了一些选择）。

如果你决定要发布缺陷数据库，那么要让你的用户知道，他们往缺陷报告里添加的信息将会被公开。

隐私问题

比尔·卡温

在一家我曾经任职的公司里，技术支持维护了缺陷数据库很多年，并一直假定它是保密的。当用户想让我们发布这个数据库时，我们不能发布，因为里面全都是一些用户的私人信息，包括姓名、电话号码、IP地址等。

4. 提供反馈

当某个用户提交缺陷报告时，需要积极回应并支持他们继续下去。

这不一定是一个繁重的任务。许多缺陷追踪系统都有这样的功能，就是每当缺陷系统的状态改变时，都会以电子邮件的方式发送给感兴趣的人。只要你一直保证及时更新缺陷追踪系统，这个功能将确保让报告了错误的人以及你认为恰当的人，都得到最新状态通知。

5. 拜访用户

当遇到真正棘手的问题，没有什么比拜访用户更加有效了。拜访用户可以比任何缺陷报告了解得更多。

双击事件

我曾经测试了一个表面上看似非常简单的缺陷。缺陷报告描述了用户界面操作的简短步骤，并声称缺陷可以被完全准确地重现。但是我尝试了很多次，也没能重现，最终只能认为“我这里运行正常”。

几分钟后，报告它的项目经理旧事重提，并让我走到他的办公桌前，在那里他向我展示了几次，完全准确地重现了缺陷。

奇怪的是，无论何时当我试图重现它时——就在他的机器上，项目经理在一旁盯着，保证我是以相同的操作步骤进行操作的——仍然就是没能重现缺陷。

最终我们找到了我们之间操作的差别，我双击鼠标的时候总是把鼠标放在同一个位置上，但是，他在两次点击鼠标时移动了非常轻微的距离。他只移动了几个像素，但是这足以导致结果的不同。

如果没有亲自看着对方使用该软件，我们是决不会发现的。

6.3 与支持人员协同工作

大多数组织会聘用那些不直接参与软件开发过程的技术人员或半技术人员。客户支持、QA（Quality Assurance，质量保证）、客户工程师、技术客户经理等都可以在调试过程中提供帮助。

您的QA团队不仅可以在缺陷出现之前帮你检测到缺陷。他们的专业知识和观点也在你努力要找到或改进重现的时候对你特别有帮助。也许你可以考虑和QA团队的一名同事在诊断阶段一同工作？

一个良好的客户支持团队能够动用他们所有的客户关系以及专业知识来凸显其在缺陷修复中的价值，尤其在帮助我们解决沟通问题方面，这个价值是不可估量的，这些问题我们在本章前面已讨论过。他们应该能够判断并确保所有相关信息被确认，得到沟通，并避免吹毛求疵以及一切无关紧要的环节。你可能会考虑要求他们执行一项特定进程，来提高此领域缺陷报告的质量（见接下来的“特性描述”）。

> **偶尔也做做客户支持。**
> ***Work in customer support occasionally.***

无论客户支持团队多么体贴地站在你的角度去和用户沟通，你也存在被用户封杀的危险。为促进与用户的亲密理解，赢得更多用户的心，表明你愿有效地为他们开发软件，你可以考虑偶尔做一下客户支持。这不仅可以帮助你了解用户，而且没有什么比这样做更能认可你的客户支持部门的同事所面临的挑战了，这样做表达了对他们能力的尊重。

QA的万里长城

由于QA团队所具有的价值主要是提供了不同的视角，因此他们需要守护的是避免与开发团队先入为主的观念“同流合污”。但是，也容易做得过火。

我曾经工作过的公司里QA团队与开发团队之间有很深的隔阂——我们甚至都不能与测试团队谈工作。唯一可以从开发团队到测试团队的信息是一个已编译的二进制文件。唯一能得到双方回应的信息就是“通过”或“失败”。

当我问创建这个组织结构的设计师时，他回答说，如果允许我们与测试团队进行交谈，我们就有可能为了通过测试来开发软件，这是“作弊”的行为。虽然理论上这有可能说得过去，但是这剂猛药却比要治的病产生的危害还要大。

特性描述

有时在把缺陷转交给开发团队诊断之前，对其进行**特性描述**很有必要。这并不适用所有情况，但可能是非常有用的，尤其对大型的项目和团队。

特性描述与诊断之间的界线是模糊的，但大致来说，这是一个**黑盒**的过程，发生在软件"外部"而不考虑其内部工作原理。与之相反，诊断是一个**白盒**过程。

特性描述的目的是要寻找缺陷"边界"。它可以可靠地重现吗？它可以发生在不同的平台上还是只在单一的平台上？多样的输入仍然会重现该问题吗？如果是，又是怎样的呢？

在下一章，我们将注意力转向心理学，一种有效的调试思维模式是什么样呢？

6.4 付诸行动

- 充分利用缺陷追踪系统。
  - 复杂度要适中，根据具体情况作出选择。
  - 直接面向用户。
  - 自动化环境和配置报告，以确保报告的准确性。
- 缺陷报告要达到如下目标。
  - 具体的
  - 明确的
  - 详细的
  - 最小化的
  - 独特的
- 与用户合作时，应该做到以下几点。
  - 尽可能地简化缺陷报告流程。
  - 沟通是关键，多为用户设身处地着想。
- 与客户支持部和QA团队搞好关系，以便在缺陷修复的过程中得到帮助。



第 7 章

务实的零容忍策略

如何能把缺陷修复与更广泛的软件开发过程相结合呢？如何估计要花多长时间去修复一个缺陷或软件中所有的缺陷呢？如何确保你的项目不会像 Brooks 在《人月神话》[Bro95]中形象地描述的那样陷入无数缺陷修复的焦油坑中呢？

在这一章中，我们将讨论以下内容：

- 什么时候修复缺陷；
- 调试的思维模式；
- 自己如何解决质量问题。

7.1 缺陷优先

一些团队会选择立即修复那些刚刚出现的缺陷（*早期缺陷修复*），而有些团队会把这些缺陷放在一边，直到开发过程完成后再去修复它们（*后期缺陷修复*）。到目前为止，早期缺陷修复是一个更好的策略。

早期缺陷修复基于两个原则。

- 那些可能发现缺陷的过程（比如测试、代码审查、让用户使用软件）要连续地贯穿于整个开发过程中。
- 缺陷修复优先于其他任何事情。

这样做的目的就是保证软件中存在的缺陷数量（包括已经找到的和尚未找到的缺陷）尽可能地少。

7.1.1 早期缺陷修复可以大大降低软件运行的不确定性

开始寻找缺陷时，你才知道还有多少剩余的缺陷需要查找。开始修复它们时，你才知道需要

花费多少时间去完成这一修复。早期的缺陷检测和修复能帮助你估算在缺陷修复上大约需要多少时间并且依此来修改你的测试计划。而后期缺陷修复模式会给你一种你一直在进步的错觉，其实你只不过是一直在积累技术债务而已——潜伏在软件表面下的一堆问题。你可能不知道什么时候能够修复它们，因为你不可能预测到还有多少问题等待你去发现。

我们可以通过图7-1看到这个过程。在项目A中（在上面），缺陷被尽可能快地发现和修复。结果，我们能测量真实的开发速度并且精确地预测我们离完成项目（消除缺陷）还有多远。对比一下项目 B（在下面），我们把测试和缺陷修复留到最后，到那个时候，我们不知道还有多少缺陷需要修复，也不知道需要多少时间来修复。我们下周会做完吗？下个月呢？还是六个月？真的没有办法确定，因为，即使我们知道还有多少要做（实际是不知道），我们也没有历史数据来帮助我们估计会花费多少时间修复。

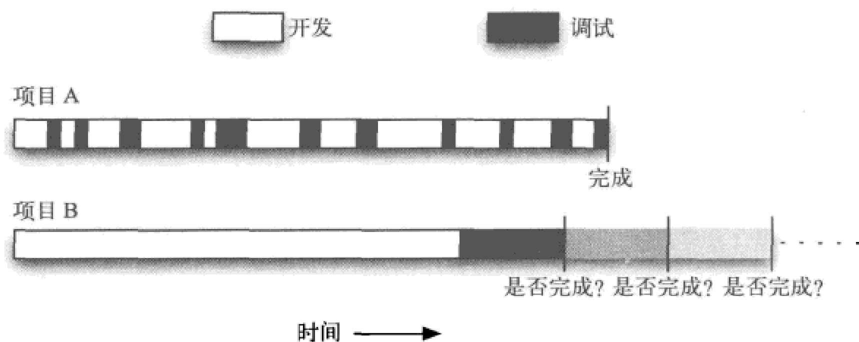


图7-1 早期检测与修复缺陷带来确定性

7.1.2 没有破窗户

编写，特别是维护软件，是一个持续的与熵抗争的过程，在这个过程中始终保持最高质量是非常困难的，这需要高水平的自律。这种自律即使是在最好的情况下也很难维持，更不用说当你面对软件有个长期没有修复的缺陷这样的事实了。只要这种自律稍有松懈，那么质量就进入一个雪上加霜的恶性循环中，这时你就真的陷入困境之中了。

> **质量低下具有传染性。**
> *Poor quality is contagious.*

很多问题具有累加效应，质量低下具有传染性，唯一确定的解决办法就是一旦发现缺陷就立刻解决它们。我们的目标就是从始到终都要保持缺陷的数量为零（或接近于零）。这种方法通常被称为没有破窗户[①]。

① 这种说法最初是在 Andy Hunt 和 Dave Thomas 的《程序员修炼之道》[HT00]一书中被用在软件方面并流行起来的。


小乔爱问……

我如何估计修复一个缺陷要用多少时间

一般情况下，是不可能估计一个特定的缺陷修复需要多少时间的。问题的诊断实质上是不确定的，只有解决了不确定因素后，所做的估计才是有价值的。

一旦你完成了诊断，就可能对修复一个缺陷需要多长时间进行一个准确的估计，但是，这对你不会有很大的帮助，因为对大多数的缺陷，问题的诊断是最耗时的。

然而，我们并没有失去什么。虽然你不能估计出修复一个特定的缺陷所需要的时间，但是可以得到关于缺陷数量的一些有用的统计数据。因此，在你发布软件的前一个阶段，如果你统计上周平均修复的缺陷数量是 20 个的话，那么有理由估计你可能在下周修复同样数目的缺陷。

早期缺陷修复利用这种原理，如果我们尽早地发现并修复缺陷，就会很快知道我们需要花费多少时间去进行调试才能得到无缺陷的软件。更理想的情况是，我们不去推测，我们只去计量花费在修复缺陷上的时间。

我们中的很多人都被“死亡之旅”项目折磨得筋疲力尽，一个成功地遵循“没有破窗户”策略的项目似乎是一个不切实际的幻想。但是，如果你在早期就发现了缺陷，并且从一开始就这么做，这样的项目还是很有可能实现的。这样，未解决缺陷的数量（包括那些你知道的和尚未发现的）将是可控的。

早日检测缺陷，从第一天就开始这样做。

Detect bugs early, and do so from day one.

7.2 调试的思维模式

我们已经看到，调试首先是一种心理活动。健康的调试心态，在工作中是很难一直存在的。有时，可能感觉好像你已经进入了《爱丽丝奇境历险记》[Car 71]的世界。

爱丽丝：再试也没用，人们不可能去相信不可能发生的事。

怀特女王：我敢说你练得还很不够。我在你这么大的时候，每天做它半个小时。有时在早餐前，我就相信有不可能的事情要发生。

天真地说，“没有破窗户”的方法也可以被理解为只有绝对完美的情况下才能实现。但是，

任何参加过软件开发项目的人都知道，缺陷是不可避免的，无论你多么努力，总会遗漏一些问题，那么，我们该如何解决这个难题呢？

一方面，我们可以认为缺陷是不可避免的，不去担心缺陷的发生，接受总会有缺陷这一事实。虽然这是基本事实，但是这种逻辑会得出一个有害无益的宿命论——不要为存在缺陷而担心，它们是软件开发中不可避免的，而你无能为力。不要强迫自己疲于实现那些不可能的事情，你只需处理那些出现的缺陷。

另一方面，作为有责任心的软件开发人员，我们的目标就是创造价值并且为此而感到骄傲，我们要精益求精。绝对不能容忍任何缺陷！不幸的是，虽然我们非常用心，如果采用这种逻辑理论，还是可能起不到什么作用，我就看到过它如何导致脆弱软件的诞生——如果从开始就没有失败过，那为什么要花时间去写一个具有故障安全功能的软件？这意味着当不可避免的缺陷确实被遗漏了，我们通常会感觉自己好像已经失败了。最坏的结果是，它会导致团队成员之间的互相指责和埋怨。



小乔爱问……

是缺陷还是特性

如果你要采取“无破窗户”策略，打算在其他开发进程之前解决所有缺陷，你可能很快就会发现自己无法判定“它是缺陷还是特性”。

从客户的角度来看，这个问题一点儿意义都没有。客户只知道软件出问题了，需要你来修复。是缺陷还是特性这个问题在客户眼里就如同“有多少天使能够在大头针上跳舞”一样无聊。

但是，从“没有破窗户”需求考虑，缺陷和特性之间的差别至关重要。你肯定不希望你精心安排的工作顺序被重新定义缺陷和特性这个小问题打乱，你也不想因为搞混了缺陷和特性而使软件质量大打折扣。

幸好两者很容易区分。缺陷是无意识行为，在此情况下软件不按照既定设计运行。除此之外都是特性，软件就是按照设计来运行的。

当然，客户抱怨的是特性并不是说就不需要改进软件，而是说问题没那么严重。

因此，如果这两个极端都是无益的，那么我们应该把目标设定在这两个极端之间的哪里呢？

> **既要完美与又要实用。**
> *Temper perfectionism with Pragmatism.*

最有成效的思维模式是务实的零容忍策略——非常接近于零容忍，但是用务实的心态去实现（图 7-2）。

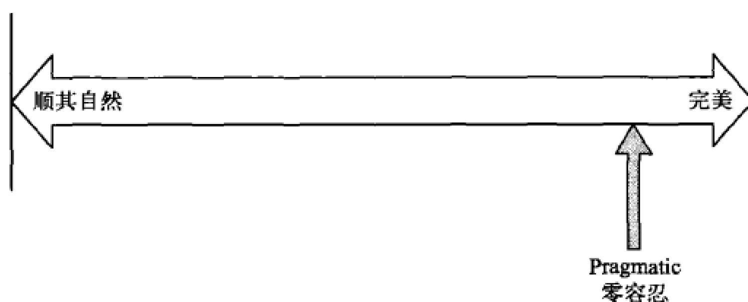


图 7-2 务实的零容忍策略

我们需要表现得好像开发没有缺陷的软件是一个可以实现的目标——不遗余力，不忽略任何可能帮助我们接近这个目标的工具和技术。当一个缺陷确实被遗漏了的时候，我们应该从这个缺陷中学到很多经验，采取一切措施确保同样的缺陷不再发生。

当我们现实地张望离想要达到的最终目标有多近的时候，我们需要做到所有这些。是的，我们要残忍地苛求自己去寻找任何可能产生的错误，但是，当我们功亏一篑要被责备的时候，不能将自己击倒了。我们需要知道，在打造一个发布伊始就很强劲的软件的过程中，缺陷是存在的，是不可避免的。

这是肯定的，缺陷一定会存在的思想会使我们有点松懈（仅仅是一点），我们远远做不到完美，但是我们可以用正确的方法无限地接近它。

7.3 自己来解决质量问题

有时，你会发现自己正面对一个包含很多缺陷的代码集。这种情况也许是你自己造成的，也有可能是你接手别人的代码时就已经这样了。不管是哪种情况都没关系。如果你面对大量的缺陷，你如何才能摆脱困境呢？

7.3.1 这里没有“灵丹妙药”

一个令人失望的事实就是，没有快速修复缺陷的方法，虽然会有一些有用的策略，但是解决问题的唯一可靠的方法就是修复所有的缺陷，当然，这需要投入大量的时间、精力，还有奉献精

神。没有任何捷径也没有任何免费的午餐。

从纯粹主义者的角度来看，一个显而易见的解决方案就是停止开发程序，并且通知其他人在你没有很好地控制质量问题前，不要进行任何新的开发。不幸的是，大多数的组织不会对你说的在接下来的六个月不发布任何新的功能产生什么好的反应。

那么，你应该怎么做呢？

7.3.2 停止开发那些有缺陷的程序

你首先要做的是设法阻止事情恶化。你可能无法立即让现有的所有代码都达到标准，但是你能确保所有新的代码都不会出现同样的错误。

> **构建实验，观察结果。**
> ***Put the basics in place.***

如果你还没有把基本要素准备好，那么第一步就应该把它们做好——没有它们，你只会让自己更深地陷入到已经置身的问题中。这意味着至少要做到以下几点：[①]

- 控制源代码；
- 一个完全自动化的生成系统；
- 一个完全自动化的测试工具；
- 可以连续构建或集成代码。

一旦这些都到位，你要确保使用它们。你在试着逆转熵，但是它们是不会被轻易解决的。改进质量要比从一开始就构建或维护它难得多。

7.3.3 从“不干净”的代码中将“干净”的代码分离出来

一个你将要去面对的挑战是，你将要面对“破窗户”效应的影响——当你被破窗户包围时，你需要坚强的意志来避免它对你的影响。

一个好的策略可以很清楚地划分“干净”（书写良好的、测试良好的、被调试过的）代码和“不干净”代码，一定确保小组中的人都知道“干净”代码的含义。

每当你有机会去做的时候，都要借助这个机会努力地了解旧代码的一些边界代码。如果你正在做这些代码，一定要编写并测试你找到的任何错误还有其他一些你接触到的问题。一段经过时间后，经过一点一滴的积累，你会发现已经成规模创建了测试，并涵盖了大量的代码库。至少在

① 我们将在第9章中详细讨论。

当前运行的所有代码区域（从质量来看，这可能是最引人关注的），最后都应当得到良好的合理的测试。

用木板挡住破窗户

在《程序员修炼之道》中，Andy Hunt和Dave Thomas曾指出，有时，你可能考虑过"用木板挡住"破窗户。

"如果没有足够的时间去恰当地修复它，那么阻断它，你可以注释掉这段你不喜欢的代码，或是标记一个'未实现'信息，或是找个虚拟的数据代替它，以防止这段代码对程序进一步的损坏，并表示你还掌控着局面。"

这种方法的一个变化形式是采用**沙箱模型**来解决问题模块。如果这段代码本身对你来说太可怕，而你没有足够的信心去修复它，那么尽可能地把这段代码从周围代码中分离出来，控制它的接口，这样你可以准确地知道它是如何被使用的，并能验证它返回的结果。随着时间的推移，你最终能够去除或重写它。

7.3.4 错误分类

在许多团队面临越来越大的缺陷数据库时，他们会选择一些特点将缺陷进行分类[①]，这种分类会议旨在构成可以查看的缺陷列表，无论新旧，都要确保你清楚地知道它们的含义和它们相互间的优先级。

> 为缺陷制定优先级别需求了解整个缺陷数据库的整体情况。
> *Prioritizing bugs requires an overview of the entire bug database.*

你会发现这些分类会议可能是最伤脑筋最花时间的了。缺陷是无限的，并且你会发现经常能够遇到在现有资源的约束下你不能权衡的错误。不过，如果你发现身处在很多没有解决的缺陷当中时，就很难能找到一个好的方法来管理这一进程。一些人需要对数据库有个概述性的了解，以便能够作出艰难的权衡，做到这点唯一的途径就是要定期更新数据库，并保证新被创建的条目有着合理的优先级。

英雄主义的价值

比尔·卡温

在一项工作中，我们创造了一个比喻后来成为笑话在流行。我们已经到了提交产品之前的最后一个礼拜了，但是仍然有一些早期标识为高优先级的缺陷没有解决。它们当

① 有时又称它为缺陷擦除会议（从清除缺陷的意义上说）。

中只有一些能够被修复。我要求大家现在去想象一下，我们的产品还有这么多错误，卡车就装载着我们“完成”的产品要离开。你是否会有强烈的感觉，我们不能让这些错误到达用户的手中，于是你要奋不顾身地冲上前去，横躺在车轮下？当然，这是一个笑话。它让我们有点参照背景，有助于让我们判断留下的高优先级错误是否都是那么重要，需要我们像个英雄一样挺身而出。在一些情况下，是这样的，但是在另外一些情况下，人们不得不承认错误还是足够的温和的。

7.3.5 缺陷闪电战

现在有一个流行的策略被一些团队采用，那就是建立缺陷闪电战（有时也被称为缺陷大集合或类似的说法）。在一个相对较短的时间内（一天、一周，也许是一个迭代周期），团队中的所有成员除了修复缺陷之外不做任何工作。

这样做的目的是在可用的时间内尽可能地降低没有解决的缺陷的数量，忽略它们的优先级。通常，这意味着一些简单的缺陷——会因为它们太不重要而被忽视掉——获得了修复的时间和关注度。

做得好的话，缺陷闪电战可以获得实际上的和心理上的双重收益，它帮助我们将缺陷的数量控制在可以管理的水平上，能帮助我们既见树木又见森林，还会让一个已经疲劳或士气低落的团队看到他们其实还在进步中。

> 小乔爱问……
>
> **我如何能重构未经测试的代码**
>
> 一旦你开始能够控制质量问题，你就想重构那些旧的、粗糙的、未经过测试的代码。你应该这样去做——关键是清除所有的问题，而重构就是这一过程的关键要素。
>
> 但是，请记住，重构非常依赖于一个广泛的自动化测试套件的支持。没有测试，你就不是在重构，而是在 hack 了。
>
> 那么，你如何重构未经测试的代码？你不要重构未经测试的代码，你第一件要做的事情就是编写测试。

然而，这是一种需要谨慎使用的技术。在短期内，缺陷闪电战可能是很有趣的——大家都在一起，缺陷的数量明显降低，公司提供比萨。但是它的乐趣只是很短的一段时间内，它很快就能

使人疲惫。我们需要感觉到自己在进步，数周的除了缺陷修复还是缺陷修复的工作会累垮每一个人。

你还需要记住，缺陷闪电战的目标就是要提高软件的整体质量。这意味着你不能在正常的过程中偷工减料——检查和权衡都是有目的的，而且都只适用于缺陷闪电战。

7.3.6 专项小组

专项小组是缺陷闪电战的一个轻微变化——一个小组在有限的时间里聚集在一起，为了解决特殊的质量问题。

如果你明确了问题域——例如，某个包含不可接受的大量缺陷的模块，这将是一个特别适用的方法。典型的专项小组，由该小组中最好的、最有经验的队员组成，他们能够找到问题产生的根本原因，拥有正确的技能和技术，能够一次性地解决问题。

在本书的下一部分，我们会看到一些需要特别留心的特殊案例，告诉我们如何建立一个帮助修复缺陷而不是阻碍修复缺陷的环境，并且避免一些陷阱。

7.4 付诸行动

- 尽早地检测缺陷，在它们刚刚出现的时候就修复它们。
- 把实现无缺陷的软件当成是可以实现的目标，依此而行动，但是既要追求完美又要追求实用。
- 如果你发现自己面临的是低质量的代码库，按照下面的去做。
  - 要认识到是没有灵丹妙药的。
  - 确保基本的要素已就绪。
  - 从“不干净”的代码中将“干净”的代码分离出来，并让它始终保持干净。
  - 使用缺陷分类，以保持你对缺陷数据库的控制。
  - 通过添加测试和进行代码重构来逐渐地优化质量低下的代码。

Part 3 第三部分

深入调试技术

本部分内容

第 8 章

特 殊 案 例

某些类型的缺陷受益于一些特殊的处理方法。在这一章中，我们将探讨一些特殊的案例。

8.1 修补已经发布的软件

所有运行良好的软件项目理所当然都会有一个发布时间规划。传统的项目可能要求每半年做一次，敏捷项目可能每两个星期做一次，但是这些项目都要发布在明确规定的、有计划的和受控制的时间点上。

这个过程不应被轻易地改变。然而，偶尔你可能会遇到一个十分严重的缺陷，你不得不打破正常的时间规划来修补已经发布的软件。

> 修补已经发布的软件的时候，要集中精力减少风险。
>
> *When patching an existing release, concentrate on reducing risk.*

诊断这种缺陷与诊断其他缺陷没有什么不同。而当你开始准备设计修复程序时，事情就会不一样起来。因为设计修补程序的目标与正常的目标有所不同。你的主要目标通常是要修复错误的根源，而对已经发布的软件进行修补时，它只是在最大程度地降低风险。

一个真正的修复可能涉及大范围的软件重构，甚至是深层次的软件体系结构的变化。在缺少对整个发布过程的正常检验和各环节平衡的前提下，很难确定这些变化将不会带来连锁反应，最终使事情变得更糟而不是更好。

因此，当实施一个补丁的时候，治标不治本的方法有时候可能是一个更好的选择。这是一个很难达到的平衡，你通常要避免“掩盖错误”。

如果你确实决定采取这种方法，那么不要认为，因为修复程序是一次“修改”，你就可以疏忽大意。与此相反，你需要更加地注意，阻止一些潜在的问题与这样的修复产生关联。虽然你不能像正常情况下对一个完整发布程序进行所有检查，但是应该进行尽可能多的检查。这个时候，

你能充分享受自己在自动化测试和发布过程中付出的努力。

> 开发版中的缺陷也需要修复。
>
> *The bug will need fixing in the development version too.*

除了修补当前版本，你还需要修复开发版中相同的缺陷。你不希望未来的某一时刻有人使用补丁版去升级软件，然后突然发现打补丁解决的问题又回来了。但是，不要盲目地作出同样的修改——开发版将经历完整的发布周期，因此，应该得到一个完善的、能够解决根本症结的修复。我们将在9.2节中讨论让你的源代码控制系统帮助你解决这个问题。

不幸的是，打补丁的时候进行了一个修复，在随后的版本中又作了另一个修复，这样会导致不同版本之间出现不兼容的行为——一个我们即将讨论的问题。

8.2 向后兼容

表面看来，定位一个缺陷是一个明确的过程。应该是这样的，但实际上，只是找到发生缺陷的原因然后修复缺陷。

许多缺陷确实一眼就能看穿。但是，有时如果缺陷出现在用户已经使用的软件中，你可能需要考虑向后兼容的问题了。

问题是，如果用户已经使用了一段时间包含缺陷的软件版本，那么他们可能依靠它在做一些错误的事。如果不考虑后果而对其进行修复，你很可能会惹恼很多用户。

当然，没有人会故意地去依赖这种不良的行为。[①]不幸的是，很容易就突然地结束对它的依赖。

- 如果缺陷影响到应用程序保存的文件，也许用户已经逐渐累积了大量损坏的文件？当在后来的升级版本中打开的时候，文件是不是可能不会给出期望的结果？或者更糟的是，根本就打不开呢？
- 如果缺陷影响你的API，那么当遇到一个修复版本的时候，任何与你的应用程序交互的代码都可能失败。
- 影响用户界面的修复可能会导致用户不得不重新学习如何操作软件（要付出相关的再培训费用）。

① 也许，要除去那些想利用软件漏洞来实现不可告人目的的人。



小乔爱问……

如果我需要一直修补现有版本该怎么办呢

修补现有的版本只适合某些特殊情况。如果它成为常规的事物，那么你就有大麻烦了。

发布补丁是昂贵的、危险的、浪费时间的。持续地这么做会导致程序很不稳定，也会使你深陷泥潭。不要坚持一个不完整的进程，需要花费时间来确定潜在的原因并修复错误。

- 也许是不同版本之间的间隔太长？考虑转向一个更加敏捷的过程，这样可以让你频繁地发布产品，或者创建一个维护计划将结构引入到维护版本中。
- 你是否有客户还"迷恋"着旧版本？你能做些什么让他们也升级吗？或许你需要使升级过程更加容易或更加可信？或去掉政治限制（例如无效升级的费用）？或者在2.0版本中重新使用那些失去的重要功能，因为这些功能是导致它们坚持使用1.4版本的原因？
- 你的问题只是简单地努力处理大量的缺陷吗？如果是这样，应该考虑采用我们在7.3节中讨论的方法。

8.2.1 确定你的代码有问题

你首先要做的就是确定你正在进行的修复工作是否可能引起兼容性问题。不幸的是，这可能会非常难办，用户可能会依靠各种各样的细微因素，对它们进行预测是很难的。

直接问他们是很难取得成效的，这种依赖关系几乎总是偶然的，因此，他们是不会有什么印象的。

> 将确定兼容性问题加入到你的缺陷修复检查列表中。
>
> *Add identifying compatibility issues to your bug-fixing checklist.*

你主要能做的就只是在你对整体了解的基础上思考一下你的修改是否会引起任何兼容性问题。为此，将它作为你的缺陷修复检查单上的一项是很有意义的，这样可以确保你不会忘记它。

你的回归测试套件可以帮助查明向后兼容性问题。不幸的是，手工建造的测试往往是简单的，运行简单的用例，相反，我们想要查明的问题是松散结合的软件区域间的复杂交互。所以，建立一个"真实世界"实例库是一个非常好的选择。你的实例库的范围越丰富，你就越容易在投入真实世界前找出问题。

8.2.2 解决兼容性问题

一旦你已经确定修复工作可能导致兼容性问题，那么你能做些什么？

你希望在两个潜在的相互冲突的目标之间找到一个平衡。一方面，针对问题进行高质量的修复。另一方面，要尽量减少因缺乏向后兼容性而引起的问题。不幸的是，同时实现这两个目标是不可能的——你可能最终会寻找一个最好的折衷方案。

你可以有很多选择。

1. 提供迁移的方法

给用户提供一些方法来修改其现有的数据、代码等东西，以适应新的规则，例如，一种转换现有的文件的实用工具，以便用户能够正确地使用新的软件。

有可能将这个过程自动化，以便在安装过程中数据可以自动升级。但要确保对其进行了认真的测试并且进行了备份——如果升级失败并在升级过程中将所有数据都毁掉，你的用户决不会感谢你。

2. 实现一个兼容模式

或者，你可以提供一个既包含新代码又包含旧代码的版本，以及它们之间的转换方法。用户可以开始使用*兼容模式*，它运行的是旧代码，当他们迁移之后再切换到新代码。理想情况下这个转换是自动的——例如，当软件检测到旧的文件。

微软的 Word 是使用这种方法的一个很好的例子。当打开一个旧文件（扩展名为.doc），它会以兼容模式运行（见图 8-1）。以新的格式保存文件（.docx），则 Word 的行为以及你的文档布局可能也改变了。

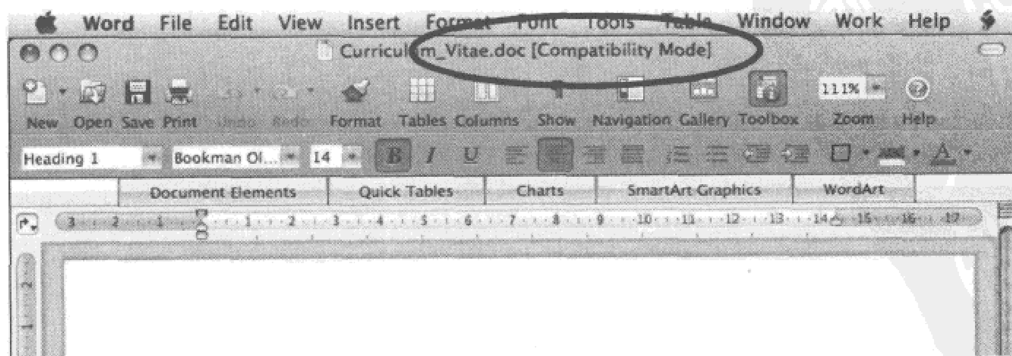


图 8-1 微软 Word 的兼容模式

兼容模式是一个非常昂贵的解决方案。

A compatibility mode is an expensive solution.

这不是一个能被轻易采用的解决办法。无论对你还是你的用户来说，它的成本都非常高。从你的角度来看，兼容模式对代码的质量没有任何贡献。从用户的角度来看，兼容模式很令人困惑——他们需要了解软件支持两种不同的行为，它们的不同之处是什么，什么时候用哪一种模式。只有付出的成本合理时才会使用这种模式。

如果你够幸运，你仅仅需要支持兼容代码一段有限的时间（可能一个或两个版本），让你的用户在这段充裕的时间内进行迁移。在此之后，你可以再次清理代码。理论上是完美的，但这些事情有一个“粘性”的习惯——只有当你成功地说服用户进行迁移时，才可以清除你的兼容性代码。当一切工作正常的时候，他们为什么会迁移呢？

3. 提供预警

如果你知道将不得不作出重大的修改，但是不必立即作出修改，就可以提供预警给你的用户，告诉他们最终需要进行迁移。例如，当Sun公司对现有Java API不满意的时候就经常这么做。

当然，只有当你能长时间延长修复以使你的用户能够迁移——并且知道用户是否真的迁移的情况下才能有效。

4. 不修复缺陷

最后的选择是把缺陷留在那里——修复相关的兼容性问题带来的代价可能比修复它带来的好处要大得多。

这不是一个好的解决方案，但是极少数情况下它也许是一个务实的选择。

这不只是你需要担心的缺陷

PostScript是一种**页面描述语言**，被用来控制打印机（手段之一）。该语言是由Adobe开发的，但很多第三方公司有自己的实现。20世纪90年代初，我就在这样一个公司工作。

当时有一个测试套件应用得非常广泛，包括数千个参考页面，占用了数米的书架空间。[①]

问题是这些参考页面都必须由真实的实现程序（在这个例子中是Adobe的）生成，

① 有一天搁板支撑崩溃了，桌子上的书架几乎要了我的命！

而且，像任何大的软件系统一样，偶尔也会出现缺陷。所以，在我们的软件生成的页面和测试套件生成的页面之间存在差异，有时这并不是我们的错。

从理论上说是这样的。

打印机的目的是打印出产品。客户对于为什么他们的页面不符合他们期望的哲学争论并不感兴趣——在那台打印机上打印得很好，那么为什么在装有你们软件的打印机上打印却会出错？因此在一些情况下我们确定，务实的做法是仿效参考实现中的缺陷。这也许并不完美，但是有时这就是世界的运作方式。

8.3 并发

并发软件是难以重现、难以诊断以及难以修复的主要问题来源。这些软件中的缺陷具有很大的不确定性，它们之间存在着微妙的并且难以理解的相互作用，并且受很多不明原因的失效模式影响。

8.3.1 简单与控制

可以在并发软件中添加很多东西帮你调试。两个关键的要素就是**简单**和**控制**。

简单是任何软件设计的关键要素之一，但是它在处理并发问题时尤为重要。保持独立线程的直接相互作用，尽可能将它们限制在一些小的代码区内，你可能会惊奇地发现可以让它们之间的相互作用如此简单。

最简单的代码也可能正常工作

我们正在设计一个服务器，在最终的配置时，将要处理成千上万的并发请求。这些线程需要共享数据，并发地访问和修改它。

共享的数据采用了树形结构，我们讨论了很长时间关于提供安全并发访问的各种方法的优点。我们制定了宏大计划，不同的子树可以锁定方便读取或写入，以及当线程同时需要多个同步锁的时候如何避免死锁带来的风险。这都是非常聪明的方法，但是有必要吗？

最后，我们创建了一个能模拟成千上万的用户访问服务器并进行负载测试的工具。结果发现，一个“多读取者，单写入者”的锁定模式比我们设想的访问方式要好得多。这大大地简化了程序——你也不会因为使用单个锁而产生死锁。

一个简单的设计不仅可以使你的软件更容易理解，并且从一开始就会减少缺陷的产生，而且也更容易控制——尤其是当你试着用并发软件重现问题的时候。如果线程只在少数明确定义的方法和少数明确定义的地方才会互相作用，那么就更容易确保在调试期间它们会以你想要的方式去相互作用。

并发软件中的大多数缺陷都会看上去非常“正常”并且与并发没有任何关系。但是在诊断期间必须处理多线程可能会使问题变得相当地复杂。因此，构建能够没有任何并发的运行软件的选项是非常有帮助的——要么将其限制为单线程运行，要么使多个线程按照定义好的顺序串行运行（取代随意的上下文切换）。

只有上下文切换发生在特定的地方和特定的时间时，大多数与软件并发有关的缺陷才能重现。可靠地进行缺陷重现取决于精确控制这些上下文切换发生的时间。正如我们在 2.5 节所看到的，有时你可以明智地使用 sleep()方法来达到这个目的，但是如果能够在同步代码中生成对于程序运行顺序的控制能力就更好了。

8.3.2 修复并发缺陷

当你要修复并发软件中产生的缺陷的时候，要记住一个关键的要素——让它们尽可能少地发生不是可取的修复办法。

通常你会发现有一个具体的“窗口”，在这个窗口可能出现竞争状态。看到如何使窗口变小可能很容易，但是要看见它如何完全关闭就并不那么容易了。

例如，你可能在几乎同一时间启动许多线程，如果发现它们的初始化代码是同时在运行，那么你的程序一定有问题。假设当第二个线程启动的时候，第一个线程已经完成了它的初始化，一个易理解的但是不正确的修复就是推迟启动多个线程。

任何这种类型修复的问题在于，如果这样的窗口没有完全关闭，早晚都会出问题。除非是在一些特殊情况下（也许是在系统负载很大，运行速度比正常情况下慢很多的时候）。你所做的一切只是使它在下一次更难重现和跟踪。

修复并发缺陷时，需避免使用 sleep()方法。

Avoid using sleep() when fixing concurrency bugs.

尤其是 sleep()几乎从来就不是正确的解决方法。我们已经讨论过，它可以作为一种极其有用的手段，强迫缺陷可靠地重现，或测试一种关于软件如何运行的理论，但是对于修复并发缺陷来说它不是合适的工具。把它当作并发编程中的 goto 语句——如果你发现自己在考虑这个问题，那么这是严重的警告了。

8.4 海森堡缺陷

海森堡缺陷，一个当你开始寻找的时候就会"消失"的缺陷，其名字来自于量子力学中的海森堡不确定理论，这个理论（不严格地说）指出在观察某一系统的过程中行为不可能不发生改变[①]。典型的海森堡缺陷确实可以重现，但是当你找寻它们的时候，它们就会隐藏起来。这会使缺陷诊断非常令人沮丧。

问题是，所有能够供你检查软件行为的技术都会在一定程度上影响其行为。不管你是通过直接在代码中进行插桩，还是在一个调试器下运行它来获得你所需要的信息，都几乎肯定会改变它的时序，它在内存中的布局，甚至两者都会改变。

对于大多数的缺陷，这不是问题，但海森堡缺陷依赖于软件某些不确定性的因素。这本身可以成为一个有用的线索。回顾 2.5 节，导致不确定性产生的原因是非常有限的，所以遇到海森堡缺陷就意味着这个缺陷一定会被这些原因中的一个所影响。

你应该进行的最快最容易的尝试是从一个收集信息的方法转换到另一个方法。如果你在调试器下运行软件，那么尝试将插桩直接加到源代码中，反之亦然。使用调试器的效果和将插桩直接加到源代码中的效果是不同的，而这种差异可能就是你所需要的。

> 尽量减少收集所需信息带来的副作用。
>
> *Minimize the side effects of collecting the information you need.*

如果你不是很幸运，那么你的任务就要变成找到某种方法来收集你所需要的信息，这种方法对软件的影响极小，保证不会改变它的行为。

日志记录是改变时序的一个主要原因——例如，调用 System.out.println()方法，需要花费成千上万的时钟周期，可能至少涉及一次环境切换。

你可以利用对不确定性区域的理解，来避免影响这些区域。例如，如果该代码包含一个紧密循环，它与另一个线程相互作用，对于循环的时序影响很有可能会改变其行为。删除任何添加到循环中的插桩程序，看看缺陷是否暴露出来了。如果暴露出来，再看看是否可以找到一种不用过多影响时序的方法来收集你所需要的资料。

内存中的日志

几年前我曾研发过一个大型的多线程产品，我发现自己要追查一个特别不易捕捉到的海森堡缺陷。我们有充足的日志文件遍布在整个代码中，并在诊断线程同步问题时多

① 这可能更多地被作为**观察者效应**而为人熟知——海森堡不确定性原理实际上涉及我们可以执行量子力学系统的准确性。但这是一个可爱的名字，所以不要太守旧。

次体现了它的价值。不幸的是，每当我打开日志功能时，代码就表现得完美无缺。

我不想失去日志功能，因为我敢肯定，它会告诉我想知道的。要是我能找到一种方法减少它对代码执行的影响就好了。

解决的办法是重新实现日志函数，而不是使用通常的输出函数，他们将日志写入内存中的缓冲区（每一个线程写一个，因此我不必担心同步访问共享缓冲区的问题）。这些缓冲区会在代码的敏感部分执行完毕后输出，随后进行交叉存取（以保证日志信息以正确的顺序输出）。

虽然新的记录功能仍会明显产生一些影响，但事实证明产生的影响是极小的，可以使问题重现了。正如我希望的那样，输出信息准确地给出了我所需要的内容，能够确定问题发生的原因。

小乔爱问……

我怎么能肯定已经修复了海森堡缺陷呢

事实上，海森堡缺陷似乎像“柴郡猫”那样容易（和令人沮丧）地时隐时现，因此确定你是否真的修复了一个海森堡缺陷是非常困难的。如果仅仅通过在调试器下运行软件或添加一个单行的输出语句就能说缺陷“没有了”，谁能说你修复的结果不是一个时有时无的行为呢？

唯一的解决办法是要比平时更仔细一些，要确保你真正地了解潜藏的根本原因。无论有什么样的疑问，宁可谨慎不可疏忽，要假设你还没了解问题的本质，不要假设你已经修复它了。

例如，你确定该缺陷是因为一个未初始化的变量产生的，并通过将其初始化为 NULL 对它进行了修复。不要停在这里，准确地弄明白，没有初始化的变量是如何引起你所观察到的行为的。它曾经取过这个值吗？如果你明确地将其初始化为这个“坏”的值，你看到了你期望看到的结果了吗？

8.5　性能缺陷

Donald Knuth 的著名论断“过早的优化是所有罪恶的根源”[①]应该深深镌刻在每一个专业软件工程师的脑海里。不管不顾地单纯追求效率比其他任何原因产生的不良代码都多。

① 摘自 *Structured Programming with go to Statements* [Knu74]。

但是，这并不意味着可以忽视效率。如果你的软件花了10分钟去执行一个本应10秒钟就完成的任务，那么可以肯定你的软件有问题。

8.5.1 寻找瓶颈

如同任何种类的缺陷，解决性能问题的关键是确定问题产生的根本原因。十之八九，这意味着你将去寻找瓶颈——代码的某个特定区限制了整体性能。

在大多数软件中，往往极少数的代码占用了大部分的执行时间。你的首要任务是确定软件的哪个部分耗费了所有的时间。这样做可以找到占用大量时间的原因。

基于这个原因，在跟踪性能缺陷时有个工具比其他工具都有效——性能分析器。

> 诊断性能缺陷之先对你的代码进行性能分析。
> Profile your code before diagnosing a performance bug.

性能分析器多种多样，各工具在工作方式的细节(例如，有些需要编译环境中有特定的钩子函数，而有些则影响未修改的代码)和产生细节信息的数量上有所不同。它们的共同特点是，在代码运行时，它们会检查代码来生成一个报告（或分析文档），说明在程序的哪部分花费了大部分时间。这是极为珍贵的数据——当你追踪到几个性能缺陷后，就会很快发现通过检查代码来预测性能瓶颈几乎是不可能的。唯一确定的解决方法就是按照从运行的软件中收集的数据运行。

因此，你的主要关注点是保证生成的性能分析文档能够准确地反映软件的真实行为。

8.5.2 准确的性能分析

适用于性能分析的观察者效应（Observer Effect）和其他任何观察代码的方法一样——至少在理论上，你观察性能的行为会影响你正在检查的程序。正是认识到这一点，这类工具的编写者都投入了大量的努力，以确保它们对分析的软件的影响尽可能地小。因此，在大多数情况下，你不必担心性能分析器本身影响你的调查结果。

更有可能对你的结果质量产生不利影响的是如何构建和运行你的软件。你需要明确以下的事情。

- 你要分析的构建应该尽可能地接近产品版本。特别是，请确保你以同样优化的水平来构建它。
- 你的运行环境要尽可能地与软件的最终目标运行环境相同。用于开发软件的机器可能也可能不符合要求，这取决于软件的类型。

- 请你使用具有代表性的数据运行软件。例如，使用小的数据集往往更有吸引力，因为它们比真实的产品数据更方便，但这可能会生成误导性的分析文件（可能会过分强调产生恒定开销的任务的影响，或者没能表现出缓存或分页机制的效果）。



小乔爱问……

如果没有瓶颈呢

有时，并不存在一个或几个瓶颈，而是软件总体运行都很慢，或者在某个不定的时间不定的地方慢下来。在这种情况下，你需要通盘地寻找哪些因素可以影响到软件的性能。主要的可能原因包括以下几点。

资源耗尽 为了满足软件的内存需求操作系统必须分页吗？你有内存或其他资源泄漏的问题吗？你正在遭受内存碎片问题的痛苦吗？

垃圾收集 如果你的软件分配很多短生命周期的对象，垃圾收集器可能要很频繁地运行。

缓存 如果你的软件实现或依赖于某种形式的缓存（内存、磁盘或其他），你遇到大量的缓存丢失的问题了吗？

8.6 嵌入式软件

调试嵌入式软件是非常棘手的，不是因为它错综复杂（尽管它可以是），而是因为它所在的运行环境。嵌入式系统通常运行在硬件上，这与你的开发环境有很大的差异，这些硬件的性能和设施都很有限，这可能造成访问必要的用来高效调试的信息非常地困难。

8.6.1 嵌入式调试工具

一些专门的工具会很有帮助。

仿真 仿真器和模拟器在精密程度和它们运行的确切细节上是不同的（例如，有的运行二进制代码就好像运行在目标硬件上一样，而有的则需要构建一个稍微不同的版本），但它们都具有相同的目标——允许你在开发机上运行和调试软件，而不必使用目标硬件。通过简化和缩短编辑/构建/测试的周期，它们可以为你节省大量的时间和精力。此外，它们提供增强的访问功能，可获得很难从产品硬件中获取的信息。



小乔爱问……

如何检测性能回归模型

性能回归可以很容易地渗入到软件中——正如我们已经看到的，通过观察来预测软件的性能是十分困难的，预测性能的变化同样如此。

因此，将性能测试纳入到你的回归测试套件中是一个很好的想法。例如，它们可以运行在具有代表性的大数据集上，如果花费的时间超出可以接受的范围就发出报告。

如果程序运行比正常情况下更快的话，这时的失败测试甚至更有价值。做出一个本不应该显著影响性能的修改后，一个测试的运行速度突然变为原来的两倍，这也可以说明存在问题。也许是你期望执行的一些代码不再执行了？

远程调试 许多嵌入式系统提供了远程调试支持。目标硬件连接到开发机上（通过电缆、网络连接或类似的方法），调试器运行在开发机上并控制着嵌入式系统。

开发硬件 一个开发电路板是为了开发而设计的目标硬件版本。它有附加接口和可能支持测试的设备，例如误差仿真。开发硬件的主要好处之一是它可以经常为电路内模仿器提供内建的支持。



小乔爱问……

它是一个硬件问题还是软件问题

开发嵌入式软件的挑战之一是软件发展与硬件发展通常是并行的。但是有一种不好的趋势是，当出现问题时做硬件的埋怨做软件的，做软件的也埋怨做硬件的。

无论对与错，通常修复一个硬件问题要远比解决软件问题更难。因此，无论是否是“你的”问题，你都很有可能是修复它的那个人。

电路内模仿器 电路内模仿器（ICE）是一个多少有点过时的术语，但一般来说ICE是一个用于将硬件和软件结合起来的调试器，以提供对嵌入式系统内部的详细访问。近来许多系统具有标准的JTAG接口，它（包括其他一些东西）提供了一种标准的手段，可以获得嵌入式硬件的调试功能。

这些工具是非常宝贵的，但它们并不总是可用的（可悲的是，支持可怜的软件开发人员

往往是硬件开发人员最后才要做的)，所以有时你可能会发现自己不得不面对原始的或不存在的调试设备。有时候，即使有这些资料，你正在追踪的缺陷也只会重现在产品硬件上。

8.6.2 提取信息的痛苦路程

鉴于设备有限，直接在目标硬件上调试问题的主要挑战往往是访问你需要的信息。但是，只要有一点想象力，你通常可以找到某种方式来与之通信。

你工作的系统控制了某些东西。你可以把这种控制手段当作通信的渠道。也许你可以使用一个液晶显示屏？或者是允许你写入数据的一个串行端口？

一个就足够了。
One bit is enough.

它不必是一个丰富的渠道，一个就足够了。你是否有一个可用的 LED？或者你可以启动的发动机？使用这种方法获得信息并不方便，但它是可能的。

将逻辑分析仪作为软件调试工具

我在开发一个打印机驱动程序。它大部分的时间工作得很好，但偶尔也会输出错误的内容。驱动程序的代码非常简单，只是把一个位图文件一点点地输送给打印机。我看不出数据是在哪里出错的。

最后，我们判断原因可能是时间问题，也许是该设备的驱动程序响应中断不够快？但怎样才可以准确测量来证实我们的理论呢？

最后解决方案是一个逻辑分析仪，一个显示数字电路信号的硬件调试工具。我修改了设备驱动程序，以提高中断例程结束时一个未使用的接口线的信号，我们将逻辑分析仪连接到该行。在提出中断时通过触发分析仪，我们可以准确地测量遭中断与成功被处理之间的间隔。

果然，大部分时间中断有充裕的时间得到处理。但是偶尔，它也会被拖延太久以致出错。通过移动设备驱动程序提交我们所在监测信号的那个点，我们可以准确地查明究竟在何处发生延误。

该解决方案是不容易的，它原来是操作系统的虚拟内存架构所导致的，需要很多的努力来解决。但至少我们知道症结所在了。

8.7 第三方软件的缺陷

独立软件的时代早已过去了。现代软件必须能够和一些不同的第三方编写的代码阵列相结

合——在某些库文件和框架上构建，从服务器接收数据，反过来也给客户端提供数据。

你早晚会遇到不是你编写的代码中出现缺陷的问题（或看似是），你不能控制它，并且可能没有源程序。处理这种缺陷给我们带来了很大的挑战。

8.7.1 不要太快去指责

第三方的代码只是代码。而且，如同任何代码，它也会包含缺陷。所以，很可能你要追踪的问题不是你自己造成的。

但是请注意——不要那么轻易地就去埋怨别人。

大部分和你打交道的第三方代码可能被用于更多的产品或有更多的人在使用。这意味着它已经受很好的测试，大部分比较明显的缺陷已经被发现。

3个月一事无成

戴夫·斯特劳斯

我们团队中一个成员为了解决一个缺陷花费了3个多月的全职工作时间。他花了很多时间试图理解一个相当复杂的第三方库文件，最后一事无成。

然后他的同事几乎只用了半天很偶然地就修复了这个问题——他应该使用受到此缺陷影响的功能，并注意到（对这个特殊的案例而言）库文件没有被正确地调用。

后来我与他交谈，他告诉我，他所做的就是假设库文件基本能够工作，基于这种假设他研究了如何使用它，答案一下子就蹦出来了。他说，假设库工作是相当安全的，因为这个代码分布地十分广泛，在许多地方都被使用。

带着怀疑的眼光对待你自己的代码。首先假设你的代码是有缺陷的。如果最后得出结论缺陷是在别的地方，那么再回来努力地看看自己的代码。当你真的已经用尽了所有的方法之后，再去埋怨第三方代码。

> 首先怀疑你自己的代码。
> Suspect your own code first.

8.7.2 处理第三方代码的缺陷

如果你已经发现在第三方代码中存在缺陷，就需要明白该怎么处置它。除了提交报告并等待编写者或厂商为你修复这个缺陷之外，你可能别无选择，但是你可以找到问题的解决办法。

或者，如果你可以访问源代码，甚至可以自己修复它。但是，这就产生了一个问题，你是否应该这样做？

报告别人代码中的缺陷

如果你在缺陷报告上做得好的话，你可以大大增加修复第三方代码缺陷的机会。换位思考一下，仔细想想你会希望看到一个什么样的缺陷报告。确保其符合 6.1 节中所描述的标准。

请记住，所有的代码都有缺陷。毫无疑问，当你花费很多时间追查问题的时候可能会很沮丧，结果，你可能对编写者没有什么好感。在沟通中不要表现你的沮丧，要有建设性，依据事实说话，这样更容易取得进展。

如果可以的话，为什么不自己解决问题？缺陷修复应该是要解决根本原因，不是吗？

> **使用你自己修补的第三方代码之前请慎重他考虑一下。**
>
> ***Think carefully before using your own patched version of third-party code.***

通常情况下是这样的。但是，第三方代码中存在的缺陷不属于通常的情况。问题是你对第三方代码所进行的任何修改，包括缺陷修复，都与别人要解决的并不一样。在团队支持方面可能会有问题，特别是当升级到新版本的时候——重新再应用自定义的修复是一个容易出错的过程，有引起回归的危险。

因此，最好的解决办法通常是在短期内暂搁问题，将你的修复融入到长期的官方发布版中。这样做的难易程度在很大程序上取决于谁拥有代码，以及你和他们的关系怎样。

8.7.3 开源代码

一种非常重要的第三方的代码是开源代码。

1. Linus法则

> **只要给予足够的关注，所有的缺陷都是浅显的。**
>
> ***Given enough eyeballs, all bugs are shallow.***

许多人认为，开源从根本上改变了调试过程。这个论点是由最著名的 Eric S. Raymond 在《教堂与集市》[Ray01] 这本书中提出的 Linus 法则——“只要给予足够的关注，所有的缺陷都是浅显的。”[①]

就 Linux 这种规模的开源项目而言，可能的确如此。但是有很多开源项目只获得了有限的关注。所以，开源并不意味着传统调试的立即结束。

① 更正式地说：“如果给予足够大型的β测试和合作开发基线，几乎每个问题可以被很快地定位，也会找到适合修复缺陷的人。”

2. 开源构建过程

迄今为止我们讨论的一切和其他任何方法一样，都适用于开源项目的开发，但是有几个特别相关的环节——构建配置和提交报告。

开源的本质意味着你不能以中心化的方式构建软件。任何人可以在他们选择的任何时间里构建它，他们构建时使用的电脑在很多方面与你的也相当不同。可能会构建在不同的操作系统上，使用不同的编译器、不同版本的库文件和不同的配置。你对此不能进行控制，但是你可以确保尽可能完整地自动化构建过程（不管谁构建了软件都不要挑三拣四），确保你能够收集到所有便于理解所需要的信息，如果有必要，可以复制它所构建的环境。

作为一个很好的例子，来看看 Firefox 的 `about:buildconfig` 页面（图 8-2）或将`-V` 命令行选项传给 Apache HTTP 服务器。

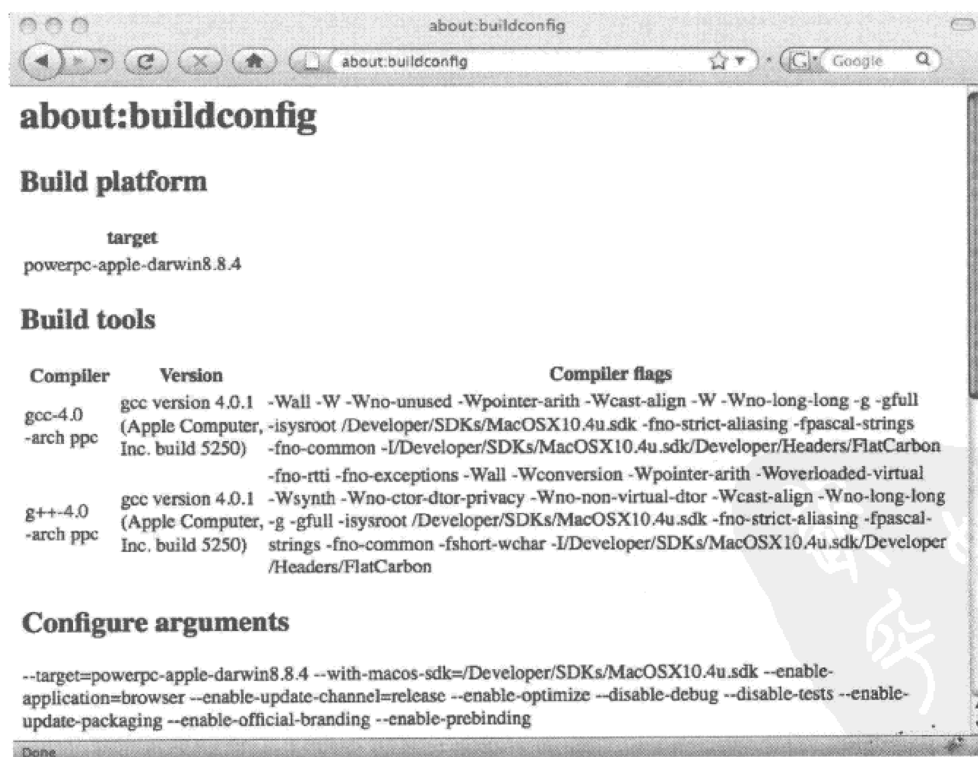


图 8-2　FIREFOX 的 ABOUT：BUILDCONFIG 页

3. 参与社区

开源社区的一个伟大之处是它可以给我们提供极大的帮助。你不仅可以得到完全免费的高质

量软件，而且往往同样高质量的技术支持也是免费的。

但是有效地寻求帮助是一门艺术。

- 首先，做你应该做的。检查文档，经常提出问题，搜索邮件列表和博客条目，看看是否有其他人遇到同样的问题。
- 尽可能多地给出信息。你已经尝试过什么了，你看到了什么结果，为什么与你期望看到的不同？
- 请记住，开源社区成员通常是志愿者。如果他们选择来帮助你（他们可能会真的帮助你），那也是他自愿的。

如果你依靠开放源码，那么要尽你所能来回报社区其他人员对你的帮助。通过报告和描述缺陷的方式参与到 Linus 法则中。如果你修复了缺陷，之后即可将修复程序提交回中央进行处理。将文档、教程和例子共享给大家，并且通过邮件列表和论坛来回答其他人的问题。

8.8 付诸行动

- 修补已经发布的软件的时候，要集中精力减少风险。
- 在修复缺陷的过程中不断监测兼容性问题。
- 确保你已完全关闭了计时窗口，而不仅仅是缩小了尺寸。
- 当遇到海森堡缺陷时，尽量降低收集信息的副作用。
- 修复性能缺陷往往始于准确的特征描述。
- 即使是最有限的沟通渠道也足以提取你所需要的信息了。
- 面对第三方代码，先怀疑自己的代码。



第 9 章 理想的调试环境

调试过程是不会发生在真空中的。提前了解好基本知识，未雨绸缪，等你真的遇到一个缺陷时，会为你节省大量的时间、精力并减少挫折感。

在这一章中，我们将讨论这些基本知识：

- 一个完全自动化的测试工具；
- 源程序控制；
- 一个完全自动化的构建系统；
- 持续构建或持续集成。

9.1 自动化测试

前面讨论过，通过广泛采用自动化测试和重构，敏捷软件开发极大地改变了软件的构建过程。我们在 4.4 节中讲述过重构，本节我们来讨论测试。

9.1.1 有效的自动化测试

有效的自动化测试不仅仅意味着简单地使你的测试自动化。为了达到效益最大化，你的测试必须满足以下目标。

明确说明测试的结果是通过还是失败　每个测试输出一个结果——通过或者失败。不要模棱两可，没有定性的输出，没有解释的必要。只简单地输出是或不是就可以了。

独立　运行一个测试程序之前不需要安装。而且在测试运行之前，它能够自动地安装任何它需要的环境，同样重要的是，测试过后能够撤销对环境所做的任何修改，恢复开始的模样。

单击运行所有测试　所有的测试都可以一步运行，而不会互相影响。与单一的测试一样，完整的测试套件的输出也就是一个简单的通过或失败——如果每一个测试都通过即认为是通过，否

则就是失败。

全面覆盖 很容易证明，对任何重要的代码来说实现完全覆盖都是极其昂贵的。但是，不要让这种理论上的限制阻碍你的工作——做到足够接近完全覆盖是有可能的。[①]

9.1.2 自动化测试可以作为调试的辅助

那么，调试的时候是什么让自动化测试更有价值呢？它们可以在各个阶段为我们提供帮助。

- 首先，经过良好测试的代码往往只有少量的缺陷。最容易修复的缺陷就是根本不存在的缺陷。
- 发生错误后越快发现错误，修复起来就越容易，成本就越低。早期测试意味着大多数的缺陷是在发生之后很短的时间内（通常是立即）被发现的。
- 自动化测试是一个持续集成的关键推动因素，在自动测试中，当代码完成之后就立即与整个产品集成起来了。在本章后面我们将进一步讨论这个问题。
- 自动化测试能使你经常发布新版本的软件，并且自信新发布的软件可以正常地运行。这意味着你可以比在其他情况下更快地从最终用户那里获得有关新功能和缺陷修复的反馈(而且，也可以在代码写完后尽快发现缺陷)。它也可以减少将缺陷修复向下移植到前几个软件版本或发布补丁的需求。
- 对于要测试的代码，它需要构建成能够获取中间结果和内部结构的方式，而其他办法可能不会实现。这种获取方法在后面的调试中被证明是有很大帮助的。
- 编写测试程序是在诊断过程中重现缺陷的极好方法。很多被用来支持自动化测试的技术对于准确地重现缺陷来说是极为有帮助的。
- 当你完成你的诊断之后，自动化测试可以提供强大的保护措施来防止在修复中引入回归。
- 如果在诊断期间，你养成了一直写测试来重现缺陷的习惯的话，那么你自然而然地得到一个回归测试，确保缺陷在将来不会被重新引入。
- 自动化测试是重构的关键推动因素，它是你所能使用的最有力的武器，可确保代码在整个开发的生命周期中始终保持良好结构和灵活性。

当与一种现在非常流行的技术——代替测试技术相结合的时候，自动化测试就成为非常强大的测试工具。

① 用极限编程的说法就是测试“一切可能起到破坏作用的代码”。

9.1.3 模拟测试、桩测试以及其他的代替测试技术

代替测试（test doubles）是在测试过程中使用"虚拟"测试对象来代替真实的测试对象。有几种代替测试技术，最常见的是模拟测试和桩测试。

模拟测试和桩测试经常被混淆。桩测试是被动的，调用时对已存的数据简单地做出响应，而模拟测试是主动的，可以验证如何以及何时被调用的预期结果。关于它们之间的区别，可以参考 Martin Fowler 的"模拟测试不是桩测试"这篇文章[Fow]。

> **模拟测试是主动的，桩测试是被动的。**
>
> ***Mocks are active; stubs are passive.***

就我们的目的而言，如果一个缺陷与系统的其他部分的相互作用是非常重要的，那么试图可靠地重现这个缺陷时，代替测试技术是十分有帮助的。

例如，我们思考一下，一个 Java 类（从一台服务器获取网络数据）用下面的接口：

```
public interface DataServer {
  boolean connect(String serverAddress);
  String fetchItem(int itemId);
  void disconnect();
}
```

假定我们在调用它的代码中发现了一个缺陷，这个缺陷仅当 fetchItem()方法在第三次被调用并抛出一个 SocketTimeoutException 异常时发生。通过择机把网线从电脑中拔出来的方法来重现缺陷不是一个有效的方式。

相反，我们可以创建一个简单地返回正确数据的桩模块来调用此缺陷：

```
public class StubDataServer implements DataServer {
  public boolean connect(String serverAddress) {
    return true;
  }
  public String fetchItem(int ItemId) {
    switch(itemId) {
      case 1: return «Data item 1»; break;
      case 2: return «Data item 2»; break;
      case 3: throw new SocketTimeoutException("Timeout from stub");
    }
  }
  public void disconnect() {
  }
}
```

请注意，StubDataServer 真的是"很笨"。如果它被用于其他方面，而不是在我们正在使用

的测试中重现这个特定的缺陷的话，它是没有任何帮助的。[①]但是，这不要紧，我们已经创建的代码只会用在这种特定的背景下，不需要在其他地方运行。

9.2 源程序控制

一个源代码控制或配置管理系统是可以跟踪源代码和整个生命周期中所有作出修改的历史信息的资源库。它也许是除了编译器或解释器之外你能使用的最重要的工具了。

从调试的角度来看，源程序控制所提供的帮助是贯穿于整个开发过程的。它是一个受控构建过程的关键要素，能确保你知道正在调试的内容，知道现在运行的代码与用户使用的代码是相同的。在诊断期间，它可以精确地定位引入缺陷的修改，帮助你跟踪所尝试的实验。当实现你的修复时，它可以确保你做出真正想要作出的修改，并且与你的持续集成服务器相对应（我们将在本章后面讨论持续集成），让它们按照预期工作。

大多数与源程序控制相关的棘手问题都是与分支有关的。分支让我们可以支持在同一个时间内并行开发某一个软件的多个版本。有两个常见原因可以解释为什么这是必需的，即稳定性和可维护性。

9.2.1 稳定性

想象一下，我们正在开发一个被广泛使用的桌面应用程序的 2.0 版。事情进展得很顺利，软件已经达到几乎可以准备发布的程度。这种情况下许多团队开始实施变更冻结——在软件进入发布生命周期的最终阶段（如α和β测试）的时候，延迟提交任何变更都有可能影响软件的稳定性。通常这意味着“只修复严重错误”。在一些项目中，这个阶段会持续数月。[②]

这很有道理，但是如果我们想启动开发一个在 2.1 版中使用的新功能，那么我们非要等到 2.0 版稳定之后再实现？

一个常用的方案是创建一个发布分支——用 2.0 版的源程序的副本。在发布之前，我们需要作出的任何修改都要进入这个分支，同时开发可以继续畅通无阻地进行。关于发布分支的图形示意，见图 9-1 左侧。

① 将它与 10.2 节“调试子系统”进行比较，在 10.2 节中我们讨论了一种使用更为广泛的技术。

② 例如，在 Firefox 3 的整个发布过程中，Beta1 版在 2007 年 11 月发布，第一个试发行版本是 2008 年 5 月发布，最终发行版是在 6 月发布的。

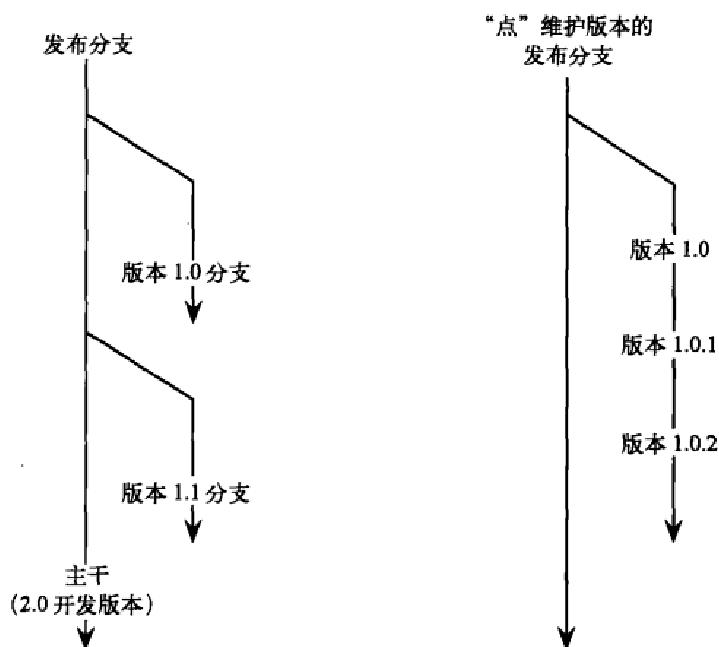


图 9-1 分支

9.2.2 可维护性

发布后运行得很顺利，许多满意的用户正在享受着 2.0 版，我们将会在 2.1 版中引入一些令人兴奋的新功能，一切进展顺利。生活是美好的。

然而缺陷报告却来了——在 2.0 版中有一个关键问题必须修复。现在该怎么办?

那么，我们当然不能从主干发布，因为我们已经对它作出了很大的修改——这些修改还没有经过足够的测试检验。这样，我们在稳定期作出的发布分支就会再次体现出它的价值，我们称之为维护分支。

我们修复着分支中的缺陷，将版本号提升到 2.0.1，生活再次变得美好。关于维护分支上点发布的图形表示，请参阅图 9-1 右侧。

这一切听起来很简单，那么分支主题为什么会使疲惫的软件工程师变得脸色煞白?

9.2.3 与分支相关的问题

分支导致重复工作。每次我们解决一个分支中的问题，几乎肯定都需要作出同样的主干改变

> 分支导致重复工作。
> Branching results in duplicate work.

(或者更糟的类似但又不同的改变)。如果我们不这样做，那么当发布下一个版本时我们将要陷入回归。如果我们存在多个活跃分支，那么在其他分支中我们也不得不这么做。

将一个分支的修改合并入另一个分支是很难理解和容易出错的，尤其是当分支有明显的分歧的时候。更重要的是，对合并后的修补程序人们总想减免测试——毕竟，我们已经在合并前的分支中测试过了，因此再次测试它的主干有些浪费时间，对吧？确实是，直到未预见的问题袭击我们的时候才不这么认为。

对于源程序控制系统的分支支持差异很大，无论是如何实现分支（有些人认为源程序好像已经被复制，有些人只不过把分支当作是一个单独拷贝的不同的“视图”），还是如何支持分支。分支可能很难理解，即使你已经使用了很长时间。[1] 接口通常会留下很大的期望，并且他们能以令人吃惊的方式与其他的源程序控制功能进行交互。[2]

因此，总会有人遗忘了那些应该被整合的变更，也总有变更未被良好整合，它们破坏了构建过程或者引入了回归。一般来说，分支往往消耗大量的时间、精力和情感。

9.2.4 控制分支

对于前面的问题来说，最好的解决办法是不要使用分支。但这并不总是能做到，有时侯使用分支是必需的。

但是大量的经验方法将有助于降低解决问题的难度。

- 尽可能晚地使用分支。可能你总想事先创造稳定分支（毕竟，如果一些稳定分支是好的，那么更多的稳定分支是不是更好呢），但是很有可能因为创建了稳定分支而使你的工作效率降低，这样是很不值得的。
- 坚持分支的单一层级。如果你发现在自己的分支上进行分支了，就应该知道遇到麻烦了。
- 设置你的持续集成服务器来构建所有正处于活跃状态的分支。
- 要经常签入一些小的修改。小的修改更容易理解、合并，必要时还好撤销。
- 只把那些确实需要在分支中进行的修改加入到分支中。

---

① 我已经记不清有多少次我一边在白板上画“分支是如何在 Subversion 中工作的”一边给我的一名同事解释——经常是几个月前就已经给他解释过的同一人。

② 例如，Subversion 的外部项目，当它们所在的项目被列入分支时它们就不会再被列入分支。

- 将分支合并到主干中，而不是从主干加到分支中。该分支代表发布的软件，因此分支中的问题要比在主干上发生的问题更严重。
- 立即将分支向主干合并。这确保了合并不会被遗忘，也确保了在你对所作的修改还有印象的时候就这样去做了。不要收集多个修改一下子全部合并。
- 保持审计跟踪，以使你知道哪些变化已经被合并，什么时候被合并的（不是所有的源程序控制系统都会自动地为你做这件事）。



小乔爱问……

还有使用分支的其他理由吗

稳定性和可维护性并不是分支的仅有正当用例——它们在探查和协作中同样是非常有用的。

如果你想安全地和一些潜在的破坏性变更打交道，私有分支可能是非常有用的。分支还可以提供一种方法使两个以上的开发人员可以针对尚未成为主系统的一部分的模块交流代码和进行协作。

为这些目标而创建的分支不同于为稳定性和可维护性而创建的分支，因为它们是临时性的，（也应该是）短命的。如果它们存在的时间过久，你应该小心了，因为它们可以通过代码提供一个“空子”而绕过你开发过程中的重要环节。

当与下一个我们将要讲到的技术——自动构建过程相结合的时候，源程序控制就会变得非常地强大。

9.3 自动构建

在调试过程中你需要控制的一个最重要的变量就是软件本身。你需要能够识别并重新创建有明显缺陷的相同软件。具体来说，你需要控制以下内容：

- 软件构建时的源程序；
- 用来构建它的工具；
- 构建时对这些工具进行的选项设置；
- 任何软件链接或附带的第三方库。

构建现代软件是一个复杂的过程，其中会使用很多需要以特定的顺序和方式来调用的工具。有些团队选择通过一个“如何构建 XYZ 项目”的文档去说明如何来做。一个好得多的解决办法就是将软件的这类知识编码为自动化构建过程的一部分。

9.3.1 一键构建

你的目标是“一键”构建过程。当一个新的开发人员可以加入团队时，你已经成功完成了构建过程，将源代码下载到一个完全没有被开发的机器中，运行单个命令，并且最终获得一个完全编写好的最终产品版本，它与团队中成员编写的程序是一模一样的。

有大量的工具可以帮助你实现这个目标。（比方说，如果你使用 Java 进行开发，那么可以用 Maven；如果你使用 C++，可用 Boost Build，详见附录 A.2 节。）如果你的软件遵循一个合理的标准体系结构，你就会很幸运，发现这些工具能给你现成的帮助。如果没有，你就不得不编写一些自定义的规则。你做这些所花费的时间会得到成倍的回报——甚至连手动步骤都是会回报很多的。



小乔爱问……

我的 IDE 构建系统怎么样呢

如果不是团队中的每个成员都使用，那自动构建系统的价值将大大降低。例如，有时开发人员可能更喜欢 IDE 为他们提供的构建系统。

大多数 IDE 都可以进行外部构建，所以看看你是否能以这种方式集成你的自动构建系统。如果不能，那么花时间使其尽可能地使用顺畅，以便能够像使用集成系统那么方便地使用它。另外，现在一些集成开发环境都支持足够成熟的构建系统，如果团队的每一个成员都乐意使用相同的 IDE，你可以以此作为自动构建的基础。

团队需要在如何构建软件上达成一致意见，并且坚持下去——如果所有开发者都以不同的方式构建，那么遇到问题是迟早的事。

自动化整个构建过程，从开始到完成。

Automate your entire build process, from start to finish.

你的系统应该始终自动化整个过程，直到最终发布软件。如果你的软件是打包成一个安装程序，构建安装程序也应该是自动的。如果在源程序控制中你使用标记来记录构建的过程，那么创建标记应当自动的。每次运行构建的

时候你不用总执行这些步骤（例如，在开发过程中在你自己的机器上构建时），但是，那只会使自动化变得更重要——恰恰是你做得越少的事越容易出错。

9.3.2 构建机器

你的自动生成过程应保证每个人都准确地得到完全相同的结果，无论他们什么时候进行构建。但是开发者的机器往往在不断变化——我们正在为新的功能修改本地的源程序，或者我们正在实验新版的构建工具。

> 永远不要在开发人员的机器上构建发布版的软件。
>
> *Never release software built on a developer's machine.*

发布软件时，任何事物所处的状态都应是众所周知的，这一点很重要。在开发机器上做到这点可能很困难而且容易出错。所以，有一个构建机器专门用于构建发布版本是一个非常好的主意（如果软件是跨平台软件，可能是几台构建机器）。它应该始终保持初始状态，不被用于其他任何事情，你可以相信它处于正确的状态。

一台构建机器是值得拥有的，但你可以通过采取进一步的措施让它变成一个持续集成服务器，为团队增加它巨大的价值。

9.3.3 持续集成

在软件开发生命周期中，有几个时间点的风险从本质上说要比其他的风险高。其中风险最大的是集成由团队中不同成员所作出的修改时。当你在本机上测试你自己的修改时，一切都很顺利，但是将你的修改与别人的修改集成起来时，就会出现问题。

> 每当你修改软件时请运行测试。
>
> *Run your tests every time you change the software.*

你的构建机器作为一个持续集成服务器实现了它自身的巨大价值。每当有任何对源程序的修改，它都会自动检查出来，构建并运行整个测试套件（一个典型的持续集成系统如图 9-2 所示）。如果构建或任何测试失败了，它会发送邮件到团队中以便问题尽快地得到解决。

有大量的非常不错的持续集成服务器包可供我们使用（A.2 节中提供了清单），但是如果有一个很好的自动构建系统，自己创造一个也是非常容易的，所以如果你需要就不要迟疑，自己做一个吧。

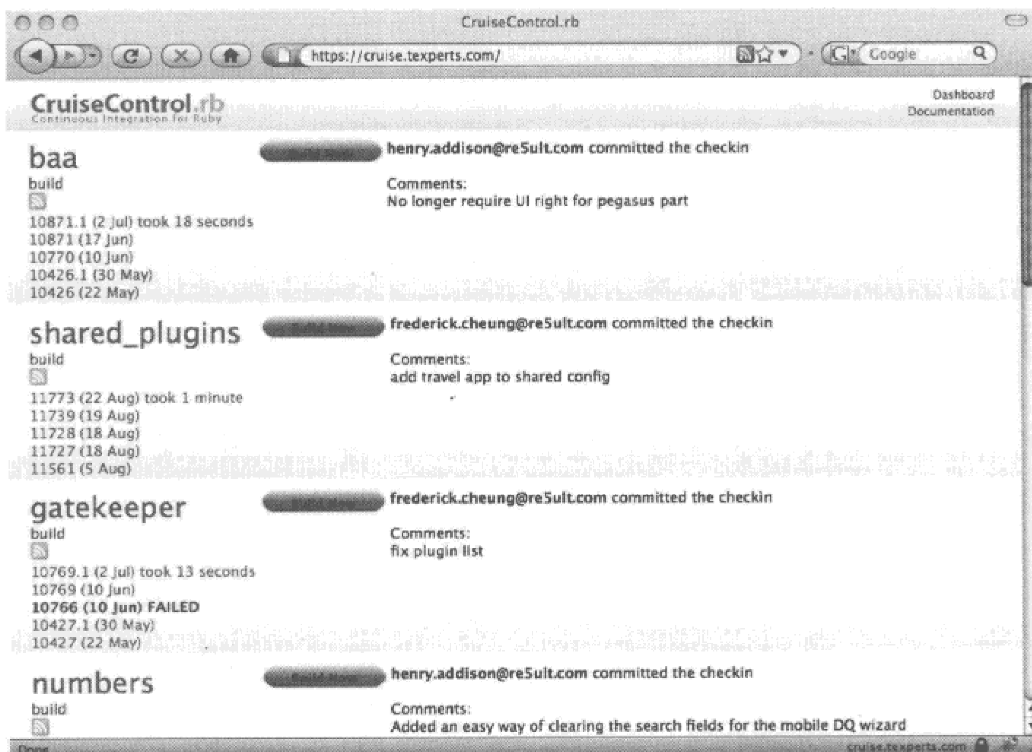


图 9-2 持续集成服务器

9.3.4 创建版本

现在，错误报告告诉你，缺陷发生在 3.6.209（e3）中。太好了。现在它要告诉你什么呢？

只有当你能够确定到底是什么对其进行构建时，它才是有用的信息，这意味着需要给源码控制程序加上版本号。因此，无论什么时候发布，你都需要确保记录了哪些源代码被用来创建发布版本。

根据不同的源代码控制系统，这可能意味着建立一个标记、一个分支、一个标签或别的东西。无论你使用什么机制，都需要让版本号与源代码之间保持一对一的关系。

> 不同的源代码，不同的版本号。
>
> *Different source, different version number.*

这是一个非常重要的必然结果——绝不重复使用版本号。如果在你刚刚发布一些东西之后就立刻发现一个严重错误，并且需要发布一个新的版本来修复它，那么要为新版本生成新的版本号。不管做出多么小的修改，下面这句话都是适用的：不同的源代码，不同的版本号。

9.3.5 静态分析

大部分的调试依赖于动态分析——在软件执行的时候检验它。但事实证明，很多缺陷可以仅通过静态地检查源代码而被识别出来。更妙的是，这种类型的静态分析可以自动化，可以被集成到你的开发流程中，在你甚至还没开始运行代码之前就检测到缺陷了。



小乔爱问……

连续构建和冒烟测试怎么样

有些团队使用连续构建和冒烟测试（以硬件工程师进行的测试来命名的，当他们第一次通上电源时，硬件是不是冒了一股蓝烟然后就不工作了呢）。微软将这种方法显赫地应用在Windows NT 的发展过程中，如 *Show-stopper!*一书 [Zac94]所述。

如你所料，持续集成、连续构建和冒烟测试有许多共同点，并有许多相同的优势。在正常情况下，持续集成是最好的，（如果你能这样连续不断地做，为什么需要通宵集成呢？）但是如果你的测试套件需要极长的运行时间，可能在晚上构建就是你唯一的选择了。

如果你的测试确实要花费很多时间而不能运行，请考虑创建一个简短的测试套件，可以检测每个签入的代码，也可以通宵地运行全部套件。

如果你已经花时间阅读过别人的代码，就会知道，一些缺陷就在你的身边。有一些特定的模式，尽管它们是合法的代码，但几乎可以肯定的是它们没有体现作者的真实意思。

下面是一个简单的 Java 例子，你可以当场指出以下代码的错误吗？

```
if(«first condition» &&
  «second condition» ||
  «third condition»);
{
  «some code»
}
```

任何曾从事过类 C 语言（C、C++、Java、C#等）工作的人，肯定都曾被上面的问题困扰过。如果你还没有遇到这样的问题，那么我告诉你，问题就是在 if 条件后的分号，这意味着后面的语句块将永远被执行，无论条件结果如何。

经验告诉我们，事实上有很多代码的模式，在一定程度上是值得怀疑的。除了上面这个例子

外，还包括无法访问的代码（可能永远不被执行，无论程序处于什么状态）和未使用的变量（被声明了，甚至被赋值了，但从来没被读取过）。例如，在下面的Java方法中，❶处的变量未被使用，❷处的代码不可访问：

```
    public static boolean allUpper(String s) {
❶     int length = s.length();

      if (s == null) {
        return false;
❷       System.out.println("Null string passed to allUpper");
      }

      CharacterIterator i = new StringCharacterIterator(s);

      for (char c = i.first(); c != CharacterIterator.DONE; c = i.next())
        if (Character.isLowerCase(c))
          return false;

      return true;
    }
```

不要忘记编译器

在开始寻找新的工具之前，不要忘记编译器。多年来，现代编译器已经具备一系列的警告信息，在某些情况下，甚至可以让专用的静态分析工具相形见绌。

问题就是它们常常不是默认启动。所以，不要仅仅因为你的代码编译器现在没有警告信息就作出这样的假设，以为编译器没有找出一些潜在的问题。花点时间去阅读你的编译器文档——通常你会发现一些有用的警告功能需要单独启用。例如，GCC编译器的-Wall选项，你可能会天真地认为将会启用所有的警告功能，但实际上却禁用了很多非常有用的功能。你可以使用-Wextra启用更多的功能，但即使是这样，仍然有很多功能需要你自己单独启用。

一个有趣的情况是，我们可能无意中编写了依靠未定义行为的代码。许多语言规范都有一些黑暗的角落，那里有可能写的是看上去非常好的代码，而事实上不可能去预测它的行为到底是什么样的。下面是一个C++的例子（C++是一门充满着黑暗角落的语言）：

```
int x = 1;
x = x++;
// x 取什么值呢？
```

答案是，x 可以取任何值——C++标准根本没有界定这段代码要做什么。在实践中，大多数编译器会做一些“明智”的事情，通常 x 将最终等于 1 或 2。但是，从理论上来说，它最终可能等于 42，该程序可能崩溃，或出现别的状况。这就是没有明确的定义。

有趣的是，如果在 Java 中编译相同的代码，那么它的行为是确定的——x 的值永远是 1。但是，不要太沾沾自喜，你这个 Java 开发人员——程序的行为可能被定义，但它仍然几乎肯定是一个缺陷。大概写这个代码的人打算让它做一些事情，而事实上它什么也没做。因此，尽管他们期望它这样做，但它没有这样做。

关于每种语言都有一定数量可利用的工具，这些工具可以检查代码寻找此类问题。它们的前身就是 `lint`，早在 70 年代就在 C 程序中发现缺陷，在某种程度上 `lint` 已经成为现如今所有此类开发工具的一个通用术语。

9.3.6 使用静态分析

静态分析的伟大之处在于它给我们提供了一种几乎免费的缺陷检测的方式。我们可以简单地通过这些工具运行代码并找到问题所在，而不是等待一个错误出现（无论是在测试过程中还是在实际应用中），然后经历漫长的问题重现、诊断和修复过程。不是这样吗？

因此，第一条规则是使用静态分析。开启编译器支持的所有警告功能，并在你的开发环境中使用任何可能有用的工具。

第二条规则是将你选择的工具或工具集紧密地集成到你的开发过程中。不要只是偶尔地运行它们，例如到了寻找某个缺陷的时候才临时抱抱佛脚。每当你编译源代码的时候都要运行它们。将它们产生的警报视为错误，并立即解决这些缺陷。[1]

> **将静态分析集成到你的构建过程中。**
> *Integrate static analysis into your build process.*

本章讲述了软件之外的一些技术。但是还有其他一些技术被构建到软件本身之中了，那是我们下章讨论的主题。

[1] 例如多数编译器会提供一个将警告视为错误的选项，例如 GCC 的-Werror。

小乔爱问……

我怎么才能没有警告呢

如果从头写一个新的项目，那么编写一个没有警告信息的代码是很容易的。但是如果从一个已经存在的代码库中开始编写，那就不是很容易了。很有可能第一次提高编译器的警告级别或运行一个新的工具的时候，你会被大量的警告信息所淹没。这些常常是代码的系统问题——你已经一而再地犯过的一些常见的错误，到现在也没有人知道，但是这些错误的每一个实例都产生了警告信息。也有一些问题是通过生成警告信息的代码“过滤”出来的（C++中的const正确性是一个典型的例子）。

解决的办法是务实一些。大多数的静态分析工具针对在哪儿生成了什么样的警告信息提供了细粒度的控制（例如，通过嵌入在源代码中的注释）。很多时候你可以通过关闭一个或两个负责主要模块的警告功能或者排除一个“问题”模块，将警告数量降低到可以管理的水平。你可以稍后回去修复其他的警告，但是在过渡期间你可以获得静态分析带来的许多好处。

在有些罕见的场合，如一个有问题的工具把合法的代码当成假警告，或者你有意地去编写“有问题”的代码，或者一个第三方库文件生成警告，上述同样的方法也会很有帮助。

9.4 付诸行动

- 自动化你的测试，确保它们：
  - 明确说明测试的结果，通过还是失败；
  - 是独立的；
  - 能够一键执行；
  - 做到全面覆盖。
- 在源代码控制中谨慎地使用分支。
- 自动化你的构建过程。
  - 每当软件发生变化时，都进行构建和测试；
  - 在每一次构建中都集成静态分析。



第 10 章 让软件学会自己寻找缺陷

已经有许多书或文章探讨如何编写好软件，但却少有讨论如何编写容易调试的软件的。

好在如果遵从创建良好软件的一般原则，即分离问题、避免复制、隐藏信息，并创建结构良好、易于理解和修改的软件，那么你也就能编写出容易调试的软件。良好的设计与调试并无冲突。

不过你仍然可以做好其他准备工作帮助你追踪问题。本章将介绍以下方法，让调试工作变得容易，甚至在某些情况下不必做调试都行。

- 使用断言自动地验证假设
- 调试构建
- 在异常处理代码中自动地检测问题

10.1 假设和断言

代码的每一块都建立在一个无数假设的平台上面——某些条件必须是正确的才能让运行结果符合预期。往往缺陷的出现是因为某些假设是不成立的或者是错误的。

避免做出这些假设是不可能的也是无意义的。但好在我们不仅可以验证它们，而且可以通过断言来自动验证。

断言是什么样子的？在 Java 中有两种格式，第一种是像下面这样的简单格式：

```
assert «condition»;
```

第二种格式包含一条信息，如果断言失败就会显示该信息：

```
assert «condition» : «message»;
```

无论使用哪种格式，无论何时被执行，断言都要计算条件。[①]如果条件计算结果为真，那么它什么也不做，如果为假，它将抛出一个 AssertionError 异常，这一般意味着程序会立即退出。

① 如果启用了断言功能，我们会很快得到。


小乔爱问……

进行单元测试的时候是否需要断言

一些人认为，对于试图使用断言来解决的问题自动化单元测试是一个更好的方法。这种想法在某种程度上可能来自于一个不幸的事实，即 JUnit 提供的可以在测试中验证某些条件的函数也叫作断言（有点儿令人费解）。

这不是一个非此即彼的问题，而是要双管齐下。断言和单元测试是解决相关但不同问题的方法。单元测试不能检测没有在测试用例中被引用的缺陷。而任何时候都可以使用断言来检测缺陷，无论是在测试中还是其他的什么场合。

一种去思考单元测试的方法就是把它们（部分地）当作保证所有的断言能够被时常执行的手段。

现在我们有足够的理论了，那么实际中该如何来做呢？

10.1.1 一个例子

想象一下我们正在编写一个需要发出 HTTP 请求的程序。HTTP 请求非常简单，由几行简单的文本构成。第一行指定方法（例如 GET 方法或者 POST 方法），一个 URI 和我们使用的 HTTP 协议的版本。随后几行包括一系列的键/值对（每行一对）。[①]对于 GET 请求这就足够了（其他的请求可能需要包括请求的主体）。

我们将定义一个名叫 HttpMessage 的 Java 类来生成 GET 请求。代码如下：[②]

```
public class HttpMessage {

    private TreeMap<String, String> headers = new TreeMap<String, String>();

❶   public void addHeader(String name, String value) {
        headers.put(name, value);
    }

❷   public void outputGetRequest(OutputStream out, String uri) {
        PrintWriter writer = new PrintWriter(out, true);
```

① 详见超文本传输协议[iet99]规范。

② 当然，如果有很多可用的经过良好调试的 HTTP 库文件，你是不会自己写这些代码的。但它对于我们学习来说是一个不错的简单易懂的例子。

```
        writer.println("GET " + uri + " HTTP/1.1");
        for (Map.Entry<String, String> e : headers.entrySet())
            writer.println(e.getKey() + ": " + e.getValue());
    }
}
```

这个类非常简单——addHeader()方法❶仅仅将一个新的键/值对添加到 headers 中，同时 outputGetRequest()方法❷生成起始行，后面依次跟着每一个键/值对。

下面是我们可能的用法：

```
HttpMessage message = new HttpMessage();

message.addHeader("User-Agent", "Debugging example client");
message.addHeader("Accept", "text/html,text/xml");

message.outputGetRequest(System.out, "/path/to/file");
```

上面的代码执行结果如下：

```
GET /path/to/file HTTP/1.1
Accept: text/html,text/xml
User-Agent: Debugging example client
```

到目前为止都是很简单的，哪块可能出错呢？

小乔爱问……

我如何选择一个好的断言消息

本书早期的一个审阅者，在 Google 北京办公室发现了一张海报，所有地方的海报都是这样，上面写着："确保你的出错消息可以帮助你调试错误，不能只提醒你需要调试错误。"

他们引用的例子是一个一般格式的断言：

```
assert_lists_are_equal(list1, list2);
```

如果这个断言失败了，那么它会告诉你两个列表中的值是不相等的。你仍将返回到代码中去寻找从列表的什么地方开始它的值变得不同。无论顺序是否改变，最好是显示出第一次出现差异的元素，或者其他一些有利于问题诊断的事物。

当然，我们的代码是非常值得信任的。但是它仅仅是显示出我们给它传递的信息。这意味着如果使用无效的参数调用，它就会生成无效的 HTTP 请求。例如，如果这样调用 addHeader()方法：

```
message.addHeader("", "a-value");
```

那么我们最终将获得如下的信息头，它发送到哪个服务器都会引起混乱的。

```
: a-value
```

通过在 addHeader()方法的头部使用如下的断言，我们可以自动地检测这种情况是否发生了：

```
assert name.length() > 0 : "name cannot be empty";
```

现在，如果我们所调用 addHeader()函数里面的参数为空字符串，当断言执行的时候，程序会立刻跳出如下语句：

```
Exception in thread "main" java.lang.AssertionError: name cannot be empty
    at HttpMessage.addHeader(HttpMessage.java:17)
    at Http.main(Http.java:16)
```

10.1.2 等一下——刚才发生了什么

让我们花一点时间来看看刚才究竟做了些什么。我们仅仅是在软件中添加了简单的一行代码，但是这一行代码的表现令我们印象深刻。我们正在教程序自己寻找缺陷。现在，当程序出现错误时，不是我们主动地去寻找缺陷，而是程序自己通知出错并告诉我们这个缺陷的相关信息。

在用户发现之前自己在测试阶段找到缺陷，这是非常理想的。但是在我们追踪现场发现的错误时断言还是非常有用的。只要我们找到一种复制问题的方法，很有可能断言就会立刻指出有什么违反了我们的假设，这在诊断中可以大大地节省时间。

10.1.3 例子，第二幕

现在，我们走上了测试的道路，我们可以走多远？其他类型的缺陷我们能自动检测到吗？

自动化地检测空字符串是没有任何问题的，但是在我们的类中是不是还有其他一些非常明显的不良的使用方式呢？一旦我们开始考虑这个问题，就可以发现很多这样的问题。

第一，空字符串不是唯一的一种无效的报头——HTTP 规范规定了许多字符不能作为报头名字。在 addHeader()函数中添加下面这些代码可以让我们确定没有使用这些关键字作为报头名：[①]

```
assert !name.matches(".*[\\(\\)<>@,;:\\\"/\\[\\]\\?=\\{\\} ].*") :
    "Invalid character in name";
```

第二，下面这些调用是什么意思？

```
message.addHeader("Host", "somewhere.org");
message.addHeader("Host", "nowhere.com");
```

---

① 不要过于担心密密麻麻的正则表达式代码——它匹配一组简单的字符。它看起来较为复杂，因为某些字符需要用反斜杠进行转义，而且所有的反斜线本身也需要转义。

HTTP 报头在消息中只能出现一次，所以重复添加就是一个缺陷。[①]这是一个通过在addHeader()函数的头部添加如下代码后我们能够自动捕捉到的缺陷：

```
assert !headers.containsKey(name) : "Duplicate header: " + name;
```

其他一些我们可以考虑的检测（这取决于我们如何预见所使用的类）可能包含如下一些。

- 确认 outputGetRequest()只执行了一次而且 addHeader()在它之后不被调用。
- 确认在每次请求中我们想要包含的报头信息都被添加了。
- 检查赋给头部信息的值以确保它们是格式良好的（例如，Accept 报头总需要被赋给一个 MIME 类型的列表）。

例子就讲这么多。那有没有一些普遍的规律能帮助确定我们该断言哪一类的问题呢？

10.1.4 契约，先决条件，后置条件和不变量

考虑在两段代码之间使用接口的方法之一是契约。调用代码保证给被调用代码提供预期一定使用的环境和参数。反过来，被调用的代码保证执行一些特定的行为或者返回一些调用代码要使用的特定的值。

考虑下面三类条件，它们一起构成一个契约。

先决条件 对于方法来说，先决条件是那些在方法被调用之前就必须满足的条件，它保证方法可以按照预期来运行。对于我们的 addHeader()来说，先决条件是参数不为空，不能包含无效的字符等等。

后置条件 对于方法来说，后置条件是那些方法被调用之后所要保持的条件（只要先决条件满足）。对于我们的 addHeader()来说，后置条件是 headers map 的大小比原来增大一个。

不变量 对象的不变量是那些只要在方法被调用之前它的先决条件被满足就始终为真的数据。例如，链表在缓存中的长度总是和列表的长度是一样的。

实现一个类时只要总按照上面这三条编写断言，你的软件就会自动检测大量可能存在的各类缺陷。

10.1.5 开启或关闭断言

我们刚刚提及的断言的一个关键方面就是它们可以被禁用。通常情况下我们在开发阶段和调

① 请注意 HTTP 规范——我知道有些时候，头信息可以合法地出现一次以上。但它们总是被能够联结值的单个头信息所代替，而对于一个简单的例子，我选择忽略这种微妙的差异。

试阶段会启用断言功能，但是在产品阶段时我们会禁用断言功能。

在 Java 中，当启动应用程序时，我们可以在 java 命令行中使用如下的参数来开启或关闭断言功能：

```
-ea[:<packagename>...|:<classname>]
-enableassertions[:<packagename>...|:<classname>]
              enable assertions
-da[:<packagename>...|:<classname>]
-disableassertions[:<packagename>...|:<classname>]
              disable assertions
-esa | -enablesystemassertions
              enable system assertions
-dsa | -disablesystemassertions
              disable system assertions
```

在其他的编程语言中，会有另一些机制来启用和禁用断言功能。例如在 C 和 C++中，我们在构建时使用条件编译来开关断言。

为什么我们要选择关闭断言功能？有两方面的原因——效率和鲁棒性。

断言的求值很花费时间，而且对于程序的功能没有任何贡献。（总之，如果程序正确运行，那么断言将不会起到任何作用。）如果断言位于性能十分关键的环节中或者条件的求值需要一些时间（想想我们之前的例子，断言参与解析 HTTP 消息是否格式良好），那么它有可能影响性能。

然而，不执行断言的一个更重要的原因是鲁棒性。如果断言失败了，软件将会随意地退出并弹出一个简短的而且（对最终用户而言）没有帮助的信息。或者说如果我们的程序是一个长时间运行的服务器，一个失败的断言没有处理完就会杀死服务器进程，谁也不知道留下的数据是什么东西。虽然在我们开发和调试错误时这些问题是可以接受的（事实上也是想让它发生的），但是这肯定不是我们在产品阶段想看到的。

相反，产品级软件应该根据实际情况编写容错性或者是故障安全机制。如何去实现这个目标不是这本书所要讲述的内容，但是这确实给我们带来了防错性程序设计这个棘手的课题。

10.1.6 防错性程序设计

> **软件在产品阶段应该是鲁棒的，而在调试阶段应该是脆弱的。**
>
> ***Software should be robust in production and fragile when debugging.***

防错性程序设计是软件开发中一个重要的且对于不同的人拥有不同的含义的术语。我们现在谈论的是通过编写运行正确（看我们如何定义正确）的代码来实现小规模容错的常见做法。

但是防错性程序设计是一把双刃剑——从调试错误

的角度来看，它令我们的工作非常困难。它把原来简单的、显而易见的缺陷转变成晦涩的、难以检测的缺陷，而且诊断起来非常困难。我们希望软件在产品阶段越健壮越好，但是调试脆弱的软件更加容易，因为当缺陷出现的时候它会立即表现出来。

一个很常见的例子就是无处不在的 for 循环，我们一般不用下面这种写法：

```
for (i = 0; i != iteration_count; ++i)
    «Body of loop»
```

我们应该书写成下面这种防错性版本：

```
for (i = 0; i < iteration_count; ++i)
    «Body of loop»
```

在几乎所有的例子中，所有的循环行为都一样，迭代从 0 开始到 iteration_count-1。因此，为什么我们许多人都很自然地编写第二种格式的循环而不是第一种呢？[①]

原因是如果循环体分配给变量 i 的值大于 iteration_count，第一种写法将不会停止循环。在测试中通过使用<替代!=，我们可以确认实际发生这种情况时循环是会终止的。

这个问题在于如果循环的索引确实大于 iteration_count，极有可能意味着代码中有缺陷。而在第一个版本的代码中，我们会立即注意到它确实有缺陷（因为程序里有一个死循环），现在换成第二个版本它就一点儿也不明显了。以后它很有可能让我们吃苦头，而且非常难以诊断。

再来看另一个例子，想象一下我们编写了一个函数，它包含一个字符串类型的参数，如果这个字符串全部是大写就返回真，否则返回假。这是在 Java 中一种可能的实现方式：

```
public static boolean allUpper(String s) {
    CharacterIterator i = new StringCharacterIterator(s);

    for (char c = i.first(); c != CharacterIterator.DONE; c = i.next())
        if (Character.isLowerCase(c))
            return false;

    return true;
}
```

这是一个十分合理的函数——但是由于某种原因我们将 null 作为参数传递给这个函数，我们的软件将会崩溃。为避免这种情况发生，一些开发者可能会在代码的开头添加一些代码：

```
if (s == null)
    return false;
```

那么，现在程序不会崩溃了——但是当程序调用函数的时候传进一个空字符串是什么意思

① 事实上，由于标准模板程式库的存在这种写法在 C++社区中是不受欢迎的，但是事实上的例子不可胜数。

呢？很有可能任何这样做的代码都包含有缺陷，但现在我们将它掩盖过去了。

断言给我们提供了一个非常简单的解决方法。无论你在哪里写防错性代码，都要确保使用断言保护这段代码。

因此，现在位于 allUpper()开头的防错性代码变成如下形式：

```
assert s != null : "Null string passed to allUpper";
if (s == null)
    return false;
```

同时我们早先的 for 循环的代码变成下面这个样子：

```
for (i = 0; i < iteration_count; ++i)
    «Body of loop»
assert i == iteration_count;
```

我们现在一举两得——鲁棒的产品软件和脆弱的开发/调试软件。

断言和语言文化

一种编程语言不仅仅是语法和语义。每一种语言周围都环绕着一个甚至几个社区，形成了自己的惯用方法、规范和实践。某种语言中是否使用断言或者如何习惯性地使用在一定程度上取决于社区的习惯。

虽然断言可以用在任何语言，但是它更广泛地应用在 C/C++的社区中。断言在 Java 中使用不广，可能是因为它们直到 Java 1.4 才被官方支持。（虽然种种迹象表明，由于基于 Java 虚拟机的语言如 Groovy 和 Scala 的推波助澜，断言在 Java 社区应用日益广泛。）

还有一部分原因可能是在 C/C++中代码出错的可能性更高。指针使用不当会出问题，字符串和其他数据结构会溢出。而这些问题不可能出现在 Java 和 Ruby 这样的编程语言当中。

但是这并不意味着断言在这些语言当中没有价值——仅仅是说明我们并不需要在这些语言中使用断言来检查这种低级的错误。在高级的问题检测当中断言仍然是极为有价值的。

10.1.7 断言滥用

和很多其他工具一样，断言可能被滥用。你需要避免两个常见错误——使用断言的副作用和使用断言来检测错误而不是缺陷。

将你的思绪拉回到我们的 HttpMessage 类上面来，然后想象一下我们要实现一个方法，用它

来移除之前我们添加的头信息。如果想要断言它总带着已有的头信息被调用，我们可能会用如下的实现方法（如果键不存在，那么 Java 的 remove()方法将返回 null）：

```
public void removeHeader(String name) {
    assert headers.remove(name) != null;
}
```

这段代码的问题是断言有副作用。如果我们在运行代码的时候不启用断言，那么它就不能被正确地执行，因为只要去掉对 null 的检查，我们就相当于移除掉对 remove()方法的调用了。

更好的书写规范是像下面这个样子（也有更好的自我记录）：

```
assert headers.containsKey(name);
headers.remove(name);
```

断言的一个任务是检查代码是否按照其应该运行的方式去运行，而不是影响代码运行的方式。由于这个原因，测试断言执行还是不执行的情况是非常必要的。如果副作用已经悄悄产生了，那么你一定想先于用户找到它们。

> **断言不是一个错误处理机制。**
>
> *Assertions are not an error-handling mechanism.*

断言是一个缺陷检测机制，不是一个错误处理机制。这有什么区别？错误是不受欢迎的，但是它们可以出现在零缺陷的代码中；而如果代码按照预期运行的话，缺陷是不可能出现的。下面这些例子几乎可以肯定是不应该使用断言来处理的：

- 试图打开一个文件却发现它并不存在
- 通过网络连接检测和处理无效的数据
- 当写入文件时空间已用完
- 网络错误

错误处理机制（如使用异常或者错误代码）才是处理这些情况的正确方法。

我们已经提到在产品版本中断言通常是被禁用的，而在开发版本或者调试版本中是启用的。但是到底什么是调试版本呢？

10.2 调试版本

许多团队发现调试版本是非常有帮助的，在帮助我们重现问题和诊断问题的很多方式上，调试版本不同于发布版本。

小乔爱问……

难道调试版本不是不同的吗

编译器的编写者竭尽全力保证，关闭最优化功能或者打开额外的检查并不改变软件的运行行为。如果你喜欢感觉好一些，编写断言和日志时也可以这么做。

但是其实调试版本不同于产品版本。在大多数的时候这无关紧要，但是请谨记在心。如果你在调试版本中重现问题时遇到麻烦，那么请用产品版本代替调试版本。

10.2.1 编译器选项

许多编译器给你提供了大量的选项，允许你精确地控制如何将源代码编译成目标代码。在开发调试阶段和产品阶段对编译器进行不一样的设置是比较合理的。下面是一些例子。

优化 现代的编译器取得了惊人的成就，生成的目标代码比手工编写机器代码更有效率、更好。然而，在生成目标代码的过程中，编译器会经常重新组织代码结构，以至于源代码和目标代码之间的关系变得越来越模糊。例如，这样做将会使在调试器中进行单步调试变得没有头绪甚至不可能。结果，调试版本使得我们无法进行优化。

调试信息 为了能在源代码中进行单步调试，调试人员必须知道如何将源代码所在位置与目标代码的相应区域建立起映射。通常这些都被排除在生产版本之外，因为它们会增加程序的大小并可能会泄露一些我们不愿透露的信息。

边界检查 一些C/C++的编译器提供对数组和一些其他的数据结构进行边界检查的功能。

但是调试版本不仅仅是选择不同的编译器选项。

10.2.2 调试子系统

有些时候值得我们去考虑特别设计一个版本来替换整个子系统，使调试更加容易。如果我们不能很容易地控制子系统的产品版的行为（例如它在第三方的控制下，或者它的运行中涉及一些随机元素），那么这种方法将会特别有用。

例如，设想我们的软件接口采用了第三方提供的服务器，我们正在试图调试一个问题，这个问题只有当这个服务器返回一个特定的结果序列的时候才会发生。我们不会很容易甚至不可能找到一种方法来确定它总能确切地返回一个满足需求的序列。就算可以，它的拥有者也不会因为我

们向服务器发出大量请求而感谢我们——特别是这些请求如果没有很好的格式（这很可能是调试过程中的情况）。

调试子系统与我们在 9.1 节讨论过的代替测试技术之间是有重叠的。所不同的是规模和范围。代替测试技术是一个短生命周期的对象，仅仅用于单一测试。调试子系统通常是用来完全替代其对应的产品子系统的，它实现了所有的接口并且在广泛的用例中正确地运行。对于我们来说，可能通过软件发布调试子系统更有意义，这样使得终端用户可以有能力在现场有序地帮助我们调试问题。

调试子系统可以完全取代相应的产品系统（模仿它的整个行为），或者将它当作软件其他部分和产品系统的中介来实施，适当地修改其行为。

在调试期间，你可能会考虑绕过一个特殊的用户接口子系统。

1. 解决用户界面问题

终端用户的需求和开发者的需求是非常不一样的。一个图形界面或者基于网络的用户界面可能会使终端用户很容易完成他们的目标，但是在开发和调试阶段却会带来麻烦，因为这样的用户界面从编程的角度来说将会非常难以控制。



小乔爱问……

如果我使用解释性的语言会如何

同样的一般原则——有的时候对于软件来说，开发和调试阶段与产品阶段的运行有所不同——也是适用的，无论是拿什么语言编写，无论语言是编译型的还是解释型的。但是，假使条件编译不能作为一个选项，这个机制对于解释性语言来说要达到目的也只有在运行时才能实现。

因为这个原因（也有其他原因），确保用户界面层尽可能地小是有道理的，仅仅是照顾显示信息的细节和征求用户的输入就可以了。尤其它不应该包含任何业务逻辑。这意味着你可以选择一个代替品，比如脚本语言，来驱动余下的软件，从调试立场上看这更加容易。

如果你的软件实现了一个对象建模，例如在 Windows 下的（OLE）自动化或者 Mac 电脑支持的 AppleScript，那就更是如鱼得水了。甚至值得专为调试增加这样一个对象模型的支持。

另一个通常被调试版本所替代的子系统是内存分配器。

2. 调试内存分配器

在C和C++这样的语言中，不提供自动化内存管理机制，因此一个调试内存分配器是非常有价值的。调试分配器可以帮助你检测和解决很多常见的问题。

- 通过跟踪内存的分配和释放，可以检测内存泄露（内存被分配但是没有被释放）。
- 在分配的内存前后放置保护器，可以检测缓冲区溢出和内存损坏。
- 通过用已知模式填充内存区，可以检测哪个实例使用的内存没有被初始化。
- 通过使用已知的模式填充已释放的内存并让其常驻，可以检测内存后又被写入的实例。

内存完整性检查器

调试内存分配器需要你对目标代码进行修改才能使用。内存完整性检查器是通过使用进程的虚拟内存结构来对任何程序进行类似分析的工具。

就个人而言，我比较倾向于使用调试分配器，因为通常其工作层级更为细颗粒化，可能给你更多的控制和洞察力。但是完整性检查器在你调试存在于产品版本的缺陷或者调试遗留应用程序时更加有用的。

附录 A.4 中列出了这些工具。

调试分配器列表可以参看附录 A.3。

10.2.3 内置控制

我们在修改第三方代码的同时，也可以选择让我们自己的代码不同于调试版本，在诊断中内置控制将会非常有用。有如下一些例子。

禁用功能 有时候你的软件包含的一些功能对于产品来说是非常有用的，但是在调试中却让人感到困惑。例如，由于安全原因，应用程序的一部分与另一部分进行的通信可能会被加密。或者为了提高内存利用率和执行速度，数据结构可能被优化。如果你选择性地禁用这些功能，那么对这些代码区域的问题进行诊断时可能会简单一些。

提供其他实现 有时实现一个模型的方法不止一种——一种是简单且容易理解的，而另一种则是复杂且难理解的。通过包含这两种代码并提供互相转换的方法，你可以证实哪种版本的结果是对的。这可以帮助你查明错误是存在于简化的版本当中还是其他地方，就算错误在其他地方，它也可以帮你更好地理解问题。

虽然我们试图讨论调试和发布两个不同版本，但是这也不能阻挡你构建其他版本。例如一些团队，在调试版本和发布版本中间编写一个半成品的集成版本。它可能启用调试标记和断言的功能使它看起来像调试版本，但是它也拥有优化功能使它看起来像发布版本。

10.3 资源泄漏和异常处理

尽你所能尽早地发现错误而不是等到它们在成品阶段才浮现出来，这个观点很对。尤其对于某类问题来说，这更是颠扑不破的真理，最具代表性的要数资源泄漏和异常处理缺陷了。

这些问题往往是相关的（资源泄漏经常是由于不正确的异常处理而产生的）和系统性的。如果你在一个地方犯错误，那么有可能在其他地方也犯相同的错误。等到问题暴露出来，你会发现面对的是一个艰巨的任务——到那时，代码会漏洞百出。

> **不要等待资源泄漏表现出来——主动且尽早地检测它们。**
> ***Don't wait for resource leaks to manifest— detect them automatically and early.***

令人高兴的是，这两个问题都可以自动检测到。在这一节，我们将看到一个用 C++编写的例子是如何来做的，同样的基本方法也适用于任何其他语言。

10.3.1 在测试中自动抛出异常

该方法是建立在两个被广泛使用的工具之上的——一个调试内存分配器和一个单元测试框架。我们将创建一个我们自己的非常简单的单元测试框架，用它来添加一个新的功能，即指出什么时候可能会抛出一个异常。每一个测试都要运行很多次。第一次是正常运行，这时测试框架只是记录会抛出哪些异常。然后再次针对每个异常运行，重新抛出这个异常。

这个方法对于任何可能抛出异常的地方都非常有用，但是有一个特别的地方非常适合使用这种方法——无论内存何时被分配。我们的例子重载了全局函数 operator new()和 operator delete()，见如下代码：

```
    void* operator new(size_t size) {
❶       TEST_ERROR(bad_alloc());
        void *p = malloc(size);
        if(!p)
            throw bad_alloc();
        return p;
    }

    void operator delete(void *p) {
        free(p);
    }
```

这段代码的关键在于调用行❶的 TEST_ERROR()代码，这段代码让测试框架知道 operator new()可能会抛出一个 bad_alloc 异常。稍后我们将会看到 TEST_ERROR()函数的实现。现在，让我们看看这段代码是如何帮我们调试异常处理的。

10.3.2 一个例子

假定我们正在编写一个用来实现简单二叉树的类。这是第一次尝试：

```
class TreeNode {
public:
    TreeNode(int value) : m_value(value), m_left(0), m_right(0) {}
    ~TreeNode() {
        delete m_left;
        delete m_right;
    }

    int value() const { return m_value; }
    TreeNode* left() const { return m_left; }
    TreeNode* right() const { return m_right; }

    void setLeft(TreeNode* left) { m_left = left; }
    void setRight(TreeNode* right) { m_right = right; }

private:
    int m_value;
    TreeNode* m_left;
    TreeNode* m_right;
};
```

这个实现本身是非常简单的——每一个 TreeNode 都有个整数值，还包含它的左右子树的指针。我们用了一些简单的获取和设置函数就成功实现了。

通过创建一个简单的测试函数，我们可以测试代码是不是按照期望的那样运行：

```
void testTree() {
    auto_ptr<TreeNode> root(new TreeNode(42));
    assert(!root->left());
    assert(!root->right());

    root->setLeft(new TreeNode(10));
    assert(root->left()->value() == 10);
    assert(!root->right());

    root->setRight(new TreeNode(20));
    assert(root->left()->value() == 10);
    assert(root->right()->value() == 20);
}
```

上面这段代码的运行结果如下：

```
Running test: testTree
   exception run: 1
   exception run: 2
   exception run: 3
```

总之，测试运行了 4 次：第一次是“正常”运行，后面三次运行每次都进行一次内存的再分配。

到目前为止，一切良好。现在，让我们更进一步，实现一个 copy()方法用来复制整个树。这应该不很难，是吧？

```
TreeNode* copy() {
    TreeNode* node = new TreeNode(m_value);
    if(m_left)
        node->m_left = m_left->copy();
    if(m_right)
        node->m_right = m_right->copy();
    return node;
}
```

看起来够简单。

让我们看看在结束测试时添加调用这个函数以后出现的结果吧：

```
Running test: testTree
   exception run: 1
   exception run: 2
   exception run: 3
   exception run: 4
   exception run: 5
Memory leaks found during test: testTree(5)
0 bytes in 0 Free Blocks.
12 bytes in 1 Normal Blocks.
0 bytes in 0 CRT Blocks.
0 bytes in 0 Ignore Blocks.
0 bytes in 0 Client Blocks.
Largest number used: 0 bytes.
Total allocations: 48 bytes.
   exception run: 6
Memory leaks found during test: testTree(6)
0 bytes in 0 Free Blocks.
24 bytes in 2 Normal Blocks.
0 bytes in 0 CRT Blocks.
0 bytes in 0 Ignore Blocks.
0 bytes in 0 Client Blocks.
Largest number used: 0 bytes.
Total allocations: 60 bytes.
```



我想实现一个异常安全版本的 copy()并不像看起来的那么容易。

当然，这个问题是如果一次递归调用 copy()会抛出一个异常，那么在方法的起始处被分配的节点就没有被删除。为了保持完整性，这里有一种修复方法，就是使用 auto_ptr 函数：

```
TreeNode* copyFixed() {
    auto_ptr<TreeNode> node(new TreeNode(m_value));
    if(m_left)
        node->m_left = m_left->copyFixed();
    if(m_right)
        node->m_right = m_right->copyFixed();
    return node.release();
}
```

10.3.3 测试框架

因此，测试框架怎样才能知道需要运行多少次测试，还会抛出哪些异常呢？框架的核心是 Test 类，它的每一个实例代表一个测试。



小乔爱问……

确定内存用尽不再是问题了

由于目前出现了虚拟内存技术，内存用尽不再是一个问题。那么，为什么要去费劲测试抛出的 bad_alloc 函数呢？既然这个问题在实际中从不发生。

问题的关键不是检查当内存用尽时代码如何处理（虽然这是一个附带的好处）。关键是检查代码是不是异常安全的。因为大部分的 C++程序都会经常地分配内存，检查我们能够处理 bad_alloc 函数是一个非常好的运行大量可能的异常处理路径的方法。

编写异常安全的 C++代码是很难的——要比它最初看起来的困难很多。对于所涉及的一些非常精彩的细节讨论，请看 Herb Sutter 编写的 *Exceptional C++*(Sut99)。

```
class Test {
public:
    Test(const char* name, void (*testFunction)());
    ~Test();

    void run();

    static bool testError();

private:
    const char* m_name;
```

```
    void (*m_testFunction)();

    void runInternal();

    // Count of errors that can be triggered
    static int m_errorCount;
    static int m_throwOnError;
};
```

Test 包括两个静态变量，m_errorCount 和 m_throwOnError。有关这些变量是如何控制测试类执行的，可以参看图 10-1。在“正常”测试运行的时候，m_throwOnError 被设置为 0，然后每调用一次 TEST_ERROR(),m_errorCount 的值就加 1。

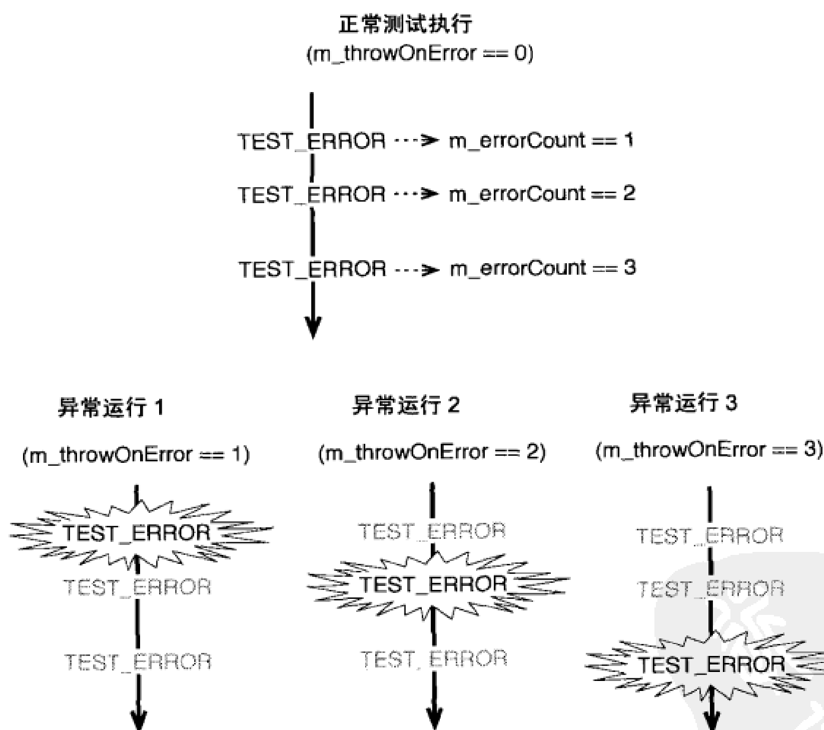


图 10-1 运行中的 TEST_ERROR()

在异常运行时，m_throwOnError 意味着 TEST_ERROR()的哪一个实例需要抛出。我们的 TEST_ERROR()宏仅仅是调用 TEST::testError()，如果它的返回值为真，那么就抛出一个异常。

```
#define TEST_ERROR(e) \
    if(Test::testError()) \
        throw e;
```

反过来，所有的 testError()方法运行结果追踪我们遇到了多少个可能的异常，如果异常的数量达到 m_throwOnError 显示的值，那么返回值就为真。

```
bool Test::testError() {
    ++m_errorCount;
    return m_errorCount == m_throwOnError;
}
```

这个是 run()方法运行的一个测试（有点让人吃惊）：

```
void Test::run() {
    cout << "Running test: " << m_name << endl;
❶   m_throwOnError = 0;
    runInternal();

❷   int additionalTestRuns = m_errorCount;
❸   for(int i = 1; i <= additionalTestRuns; ++i) {
        cout << "   exception run: " << i << endl;
        m_throwOnError = i;
        runInternal();
    }
}
```

❶ Test::run()开始于调用 runInternal()，并将 m_throwOnError 变量设置为 0 来确保 testError()方法总返回假。

❷ 在 runInternal()运行完成以后，m_errorCount 变量是我们在进行复制操作时这部分测试可能会抛出异常的次数。

❸ 然后在每一次可能的异常出现时调用一次 runInternal()，并将 m_throwOnError 设置成我们这次想要抛出的异常的编号。

最后在检测完内存泄露和不可预计的异常打包检测以后，用 runInternal()方法简单地调用测试。

```
void Test::runInternal() {
    m_errorCount = 0;
    takeMemorySnapshot();

    try {
        (*m_testFunction)();
    } catch(exception& e) {
        // An unhandled exception is only a problem if this is a normal
        // run - we expect unhandled exceptions during error simulation
        if(m_throwOnError == 0)
            cerr << "Unhandled exception in test: " << m_name << "\n" <<
                e.what() << endl;
    }
```

```
    reportMemoryLeaks();
}
```

因此，最后你将得到的是完全自动检测内存泄露和异常非安全的代码。不是这样吗？

10.4 付诸行动

- 使用断言来做下面的事：
  - 记录和自动验证假设；
  - 虽然在产品发布的时候要有鲁棒性，但在调试期间要确保软件是脆弱的。
- 创建这样的一个调试版本：
  - 以调试友好的方式进行编译；
  - 允许关键子系统被等价的调试代码取代；
  - 内嵌在诊断阶段非常有用的可控性。
- 在问题出现之前抢先检测系统问题，例如资源泄露和异常处理问题。



第 11 章 反模式

我们都熟悉模式，它用来解决常见的、反复出现的问题。

反模式是一种另类的模式，指我们反复犯的一些常见错误。有时候，它们貌似很好的解决方案，但在实践中却行不通。有些我们明知道它们不是解决问题的好方法，但却都在使用它们。

有备方能无患，了解反模式是避免发生错误的第一步。

11.1 夸大优先级

我早年所在的团队遇到一个问题。作为惯用的方法（至今仍如此），我们使用一个缺陷跟踪系统，对每一个缺陷都用数字标出其优先级。我们的优先级从 1 到 4，1 级是一些不严重和影响有限的缺陷，4 级是那些可能引起系统崩溃的、需要优先解决的缺陷。这么做似乎很不错。

然而，我们有太多缺陷需要修复，结果我们只会注意那些最高优先级的缺陷。当然，人们很快就会发现，如果不将一个缺陷设置为最高优先级，就几乎不会被人理睬。因此，没过多久，我们几乎把数据库中的每一个缺陷的优先级都设为了 4 级。

这是一个问题，因为所有缺陷的重要性几乎都是一样的，我们该如何才能知道哪些是真正需要最先解决的呢？

我们于是创建一个新的优先级，5 级，专门针对“真正关键”的缺陷，这种方法实行了一段时间。但是，你或许也看到该方案的缺点了——过了一段时间，我们又和原来一样了，只不过这次所有的缺陷级别都被设为 5 级。

我离开的时候，我们已经把优先级提高到 7 级了。

解决方法

如果你发现自己面临着夸大优先级的问题时，那么可以采用如下的解决方法。

- 定期清除你的缺陷。控制缺陷数据库——定期进行审查，并确保缺陷的重点优先级确实能够反映出它们真正的优先级（代表着修复它们会给组织带来的价值）。
- 控制缺陷的优先级。允许用户指定严重性，而不是优先级。要有一个良好定义的过程，通过这个过程来分配优先级（例如，一个类选团队）。
- 不要用数字来表示优先级，按照优先顺序把缺陷列出来。这很类似于 Scrum 推荐使用的产品订单方法（见《Scrum进行敏捷项目管理》[Sch04]）。

这些解决方案中没有一个能够找出发生缺陷的根本原因——质量差才是导致大量缺陷的原因。如果缺陷的数量不断增加，只是集中于管理缺陷的方法是治标不治本的。

虽然这不是一件容易的事，但惟一的、真正的解决办法就是控制产品质量。

11.2 超级巨星

我曾经和一个“巨星”共过事。他是团队的核心人物，是一个在紧急关头大家可以依靠的、能够提出解决方案的人。他非常聪明，工作效率也很高，远远超过团队里的其他任何人，他对整个工作流程了如指掌，可以承担任何任务。

可想而知，管理层非常喜欢他。要是我们都成为他的克隆，我们所有的问题都将得到解决。

他写的代码中偶尔也会有几个问题，但这些小问题可以由普通的工作人员轻松地处理，而他得去解决下一个难题。

但愿如此。

虽然从表面上看，他有着令人难以置信的高效，但是对他的代码进行一个粗略的检查就会发现无数的问题显现出来。显然他的代码是匆忙写就的，设计没有经过深思熟虑，没有进行充分的测试，存在不必要的重复。结果就是大量的缺陷，遍布在他刚刚实现的新功能和其他老地方。

他这么高产现在看来真有点莫名其妙——他只做了一半的工作。同样莫名其妙的是其他人的生产效率非常低下——他们花费所有的时间去清理他留下的烂摊子。当然，他没有承担任何责任，因为当问题出现的时候，他早已经离开了，着手解决下一个引人注目的、需要的“巨星”才能解决的难题了。

> **巨星效应破坏了团队。**
> ***Prima donnas destroy teams.***

如果一直使用的话，这个反模式特别具有破坏性。它会发出错误的信息。小组成员会觉得有责任心会适得其反。简单易行的变通方案得到了喝彩——忘记质量天地宽。总有其他一些可怜虫能够进行收尾工作。

当然，这些所谓可怜虫是不可能感受到工作的乐趣的。他们的士气会受到影响，这会导致工作质量下降（怕什么？显然没人关心）或大量员工离职。

解决方法

恃才者傲物。他们能够而且应该成为一个非常有价值的团队成员。关键在于如何驾驭他们的才能。

之所以有人目空一切是因为他们可以这么做。要确保你的开发过程包含足够的检验机制和平衡机制，而且人们不能做了错事而不受处分。

- 确保“完成就是完成。”在认真仔细地完成当前工作之前不允许任何人进入下一个任务。这意味着所有的功能都被测试过了、检查过了、记入文档了，你的开发过程要求的任何环节都进行了。
- 将大任务分解成具体的小任务。对待每一个任务要么“完成”要么“没完成”——不要模棱两可。五个项目的 80%完成了等于什么也没完成。四个项目做完了，一个还没开始做，这才叫 80%完成了。这给了你一个很有效的方法来告诉你真实的进度。
- 采用“自负自责”的原则——解铃还需系铃人，谁造成了缺陷谁就修复它。如果团队的核心人员产生的问题后期才暴露出来，他们应该停止一切已经转做的新工作，不管这项工作多么重要，回过头去找到问题所在。如果他们现在做的工作太重要了，不能因清理自己的烂摊子而暂停，那么就让其他人来进行新的项目，而不是去帮助他们擦屁股。

> 小乔爱问……
>
> **如果我不负责会怎样呢**
>
> 本章的一个早期审阅者曾经问：“这些主意不错，但如果不是由你（读者）来决定，管理层不这么做该怎么办？”
>
> 其中的一些过程可以通过“草根”运动引入，这样带来的压力会出奇地有效。但如果它不起作用，任何有权力的人都不采取行动，那么有时你该考虑拍屁股走人了。

11.3 维护团队

一些组织选择将开发团队和维护团队分开。开发团队开发软件，然后，一旦准备部署，就将其交给维护团队，由维护团队负责缺陷修复和运行过程中需要做的改进。

如果你一开始就认为，开发软件所需的技能不同于维持软件所需的技能，那么这样安排工作似乎有点道理。不巧的是，这种结构有很多问题，会带来一系列的问题。

- 首先，开发软件所需的技能与维护软件所需的技能没有什么大的不同。软件工程就是软件工程，不管你是新建项目还是改进已有的项目。
- 确定你的设计是否符合实践，唯一的办法就是看它在实践中的运行效果。有些问题只有当真实的用户使用软件的时候才会暴露出来。这些问题由原始的设计者进行修复更为有效，因为他比其他人更了解软件的代码。
- 与上面说的有关，如果设计师已经转向其他工作了，那么他们如何知道自己的工作成功与否？如果他们不打算反复再犯同样的错误的话，学习这些经验教训是十分关键的。
- 虽然你的项目计划可能需要清晰地将开发阶段和维护阶段分开，但现实可能是非常不同的。大多数软件在维护阶段要比在初始开发阶段有更多的工作要做。这么说有充分的理由，经常是当你向用户提交软件的时候，他们才意识到应该需要什么。
- 你可能认为随软件一起发送的那部分就是维护团队需要的所有信息了。实际上，这几乎是不可能的，不只是因为当时间进度比较紧的时候会首先压缩文档工作。关于软件的很多知识不可避免地都是隐性信息，这些隐性信息很难从文档中获得，不管你多么一丝不苟。[①]
- 维护团队几乎总是成为二等公民。由于这种二等地位的感觉，能力强的开发人员往往会进入开发团队而能力弱的开发人员会进入维护团队。这就导致了“他们是他们，我们是我们”的情况。开发团队的人不明白为什么那些维护团队的人为何无法顺利地运行软件——毕竟，最难的工作已经完成了。维护团队的人不明白为什么那些开发团队的人提供了一大堆毫无价值的信息却不用承担任何责任。
- 如果创建软件的团队首先知道，他们将来也要维护这个软件，那么团队成员将有动力确保软件尽可能地易于调试和改进。正如我们已经看到的，从一开始，就可以预设好很多事情来帮助我们。但是，如果是其他人负责，那么就会不可避免地把它排在所有要做的工作的最后面。

这种反模式也适用于个人开发人员。让一个新的团队成员加入到修复缺陷中，对他熟悉这个项目来说是一个良好的、温和的开端。但是，如果让他们只进行缺陷修复，从长远的观点看，不但对于他们而且对于其他成员都是没有任何好处的。

① 对于在团队内部及团队之间如何进行交流，参看《敏捷软件开发：合作博弈》[Coc06]。

解决方法

> 从最初的构思到最终布署，整个过程用同一个团队。
>
> *Keep one team from initial concept through to deployment.*

从最初的构思到最终布署的整个过程乃至以后的维护都要让一个团队来完成。这样可以保持连续性，确保团队成员的工作重点与组织的工作重点紧密关联，并允许他们在软件的生产过程中学习如何维护软件。

注意“专项小组”（见7.3节）不是我们这里讨论的维护团队。专项小组是为解决特定的问题临时组成的实体，它不会在组织结构中长期存在。

11.4 救火模式

救火模式是一种行为模式，在这种模式中，面对很多关键问题，我们辗转于其间，应接不暇地解决一个个严重的问题。

我们所有人都有这种倾向。当客户、管理层或同事都冲着你大叫，而最后期限将至，你走投无路，别无选择。在极少的情况下，它可能是一个合适的行为——有时你真的只做了必须做的，刚好把问题立即解决掉。

但是，如果你经常或长时间地陷入救火模式不能自拔，这就是一个大问题了。

解决方法

长期或反复地救火会破坏代码质量和团队士气。如果你发现自己陷入了救火模式，不妨退一步，先找出问题产生的根本原因，再去解决这些问题。

说起来容易，做起来难。你没有时间去寻找问题产生的根源——你已经把所有的时间都花费在解决一个又一个问题上。这种关头以退为进并着眼于全局，是非常困难的。

> 救火模式永远也不会修复任何质量问题。
>
> *Firefighting will never fix a quality problem.*

不幸的是，没有一次救火行动会解决你的质量问题。事实上，恰恰会适得其反。

如果你在一周之内一直采用救火模式，局面却仍然处于失控状况，只是更加努力地工作是不会也不能有什么效果的。无论得到什么样的短期结果，你都需要停下来，让自己找出并修复根本原因。

这可能意味着你要采取一些令人不快的决定。您可能需要承受一些短期的痛苦，面向长远打好坚实的基础（7.3节给出了如何做的建议）。

11.5 重写

当面对一个特别麻烦的软件时，使用亚历山大大帝的方法来解决问题是很有诱惑力的——斩断戈尔迪乌姆结，丢弃现有代码而快速地解决问题，然后重写新的代码。

有时候，这确实是正确的解决方法，但经验表明，我们的软件工程师喜欢动辄就使用这种方法。

从心理学的观点来看，开发新的代码比修改旧的代码让人更加舒服。我们天生的乐观感使我们低估了复制旧功能所要付出的精力和时间。

老旧生锈与全新铿亮

在软件以外的世界中，我对赛车充满兴趣。因此，我花费很多周末的时间修理我的赛车，进行日常维护，修复最近发生的损伤，或者为了在比赛中再赢得关键的一点时间而升级零部件。

由于多种原因，有次我从头开始构建了一辆新车。相对于以往，这是一段美妙的经历。我不用去修理那些可能几年前就松开了的、如今凝结在油污中的螺母和螺钉，而是直接使用能够很好地协调工作的新元件。要是每次都是这样该多好。

但赛车永远不会"即插即用"。起初的几场比赛，你不断寻找和解决初期必然发生的问题——那些小毛病导致车速下降，或者让你中途退出比赛。只有在解决了所有这些问题，你才能充分发挥汽车的全部潜力。

在这个领域中取得领先地位的赛车手，都是那些坚持不懈、始终如一地一点点提升赛车的质量的人。

即使代码事实上并没有很好地构建，没有很好地测试，或者没有很好地记入文档，即使它作为产品只有很短的一段时间，它也是大部分时间在工作着。这就意味着，它蕴涵与问题有关的大量知识——这些知识不可能在其他的任何地方获得。

这种知识是微妙的、难以在需求分析过程中重新获得的。产品阶段出现的特例——"是的，它通常应该那么做，但对这一特定类型的记录，它应该有不同的行为"——可能除了在源代码中之外，不会被记录在任何文档或其他地方。重写该软件，除非你非常小心，否则就会陷入回归问题，让以往的经验教训再来一遍。

解决方法

对任何重写的建议都要抱着怀疑的态度。进行一次非常细致的成本/收益分析。有时，旧的

代码真的是烂到了家，我们不值得再使用它，但应花些时间来证明给自己看。

如果你决定走上这条道路，那么尽可能减小风险。试图找到一个方法来增量地重写代码，而不是彻头彻尾地推倒重来。

> **避免彻头彻尾地重写。**
> *Avoid "big bang" rewrites.*

对现有的代码进行测试，并验证得到了相同的结果。要特别小心地找出现有代码能正确处理的和你需要重做的界限。

11.6 没有代码所有权

极限编程的一个做法（见《极限编程解释：拥抱变化》[wCA04]）就是集体代码所有权，每一个团队成员对所有的代码都负有责任。尤其是，任何人都可以在任何地方修复任何缺陷而不必与原作者进行沟通。

极限编程的流行（或者说肆意横行）使得很多团队采用了这样的方法，并不总是应用在极限编程的框架内。这可能会导致一些问题。集体代码所有权可以非常地有效，但应用不当，很容易沦为一种情况，即变成没有代码所有权了。任何人都可以在任何时间修改他们想要修改的任何东西，导致质量低劣，甚至落得一塌糊涂，其中代码被来回地重构，谁看了都想改一改。

解决方法

在极限编程中集体代码所有权是有效的，因为它有很多其他的极限编程方法支持，特别是结对编程、测试优先开发，以及统一的编码标准。如果没有这些或者其他的实践来提供类似的支持的话，采用集体代码所有权就很危险。

如果你不能采取这些辅助手段，或许共享代码所有权并不适合你。考虑采用一个传统的模式，让团队成员（包括大团队中的小团队）每个人都拥有一个模块。

11.7 魔法

你不会觉得我们软件工程师会很迷信吧。软件可能是与你工作在一起的最透明的实体——如果你想知道为什么它的表现会是那样，那么你需要的所有材料都在源代码中。

然而，许多项目似乎有自己的一点点魔力。

- "是啊，出于某种原因，在那台服务器上构建的程序总是出错。不知道为什么，只要你确保始终不要从那台服务器构建程序就可以了。"

- "哦，你出现了那个错误。你需要确保按照正确的顺序启动软件。它不应该有差别，但由于某种原因它确实会有差别。"
- "是啊，第一次总是失败，但之后总是完美收官。不要担心。"

问题是这种话本身就表明你不能理解软件的某些方面。任何你不理解的事物都可能隐藏有缺陷。

解决方法

在这种情况下，唯一的解决办法的是纪律。将你不理解的任何事物都当作缺陷。即使经过调查，你确定它不是缺陷，那你也一定会学到一些东西。

> 把你所有不明白的东西都当作缺陷。
>
> *Treat anything you don't understand as a bug.*

本章涵盖了一些常见的反模式。你知道，我们没有列全。人类的聪明才智正体现在，我们已经发明了很多新方法让自己的生活更加狼狈。要用挑剔的目光来持续地检查你的过程和结构，确保它们真正地使你更接近目标。

在你的软件工程师的职业生涯中，你会遇到让人沮丧、让人愤怒、晦涩的和有时十分怪异的软件。我希望已经介绍的工具、技术和方法会给你一些帮助和启发，让你意识到自己最终将取得胜利。一分耕耘一分收获，收获的喜悦将回报你所有的辛劳。祝你一帆风顺！

11.8 付诸行动

- 控制缺陷数据库，确保它准确地反映了缺陷的优先级。
- 自负自责——任何人在完成当前任务之前都不允许转向其他任务。如果在他们的工作中出现了缺陷，让他们自己来修复。
- 从最初的构思到最终布署的整个过程及以后的维护都要让同一个团队来完成。
- 救火模式永远不会修复质量问题。花些时间找出并修复根本原因。
- 避免彻头彻尾重写。
- 确保你的代码所属权策略是清晰的。
- 将不理解的任何事物都当作缺陷。

附录A 资源

这可能是老生常谈，但也能很好地解释“如果你只有一把锤子，一切看起来像一个钉子”这个谚语存在的理由。专业人士的标志就是了解哪些工具可以使用并能够选择合适工具完成任务。本附录指出了一些使用较为广泛的工具。

A.1 源代码控制及问题追踪系统

由于有庞大的范围可供挑选，选择一个适合你的源代码控制和问题追踪系统并不困难。那么，什么可能动摇你的决定？需要考虑的事项包括以下几点（并不全面）。

- 开源的还是商用的？
- 你需要自己管理它（例如，在你的防火墙之后），还是想托管给某一个服务商？
- 你是否需要将源代码控制系统和问题追踪系统紧密地结合在一起？
- 你需要对分布式开发有多大的支持？

这里我无法全面分析所有不同的源代码控制和问题追踪系统，但我可以提及几个主要工具，告诉你为什么要考虑使用它们。

A.1.1 开源解决方案

CVS http://www.nongnu.org/cvs/

直到不久前，唯一真正开源的选择还是 CVS。然而，CVS 存在一些众所周知的限制，尤其是 check-in 操作不是原子性的，并且它不支持版本目录结构。

Subversion http://subversion.tigris.org/

在过去几年，CVS 几乎被 Subversion 完全代替。Subversion 设法解决了 CVS 大部分明显的缺点，并且已经成为默认的开源选择。

Git http://git.or.cz/

某种程度上，真正发展迅速的是 Git，Git 是从大量受到高关注度的项目中获取思路转化而来，这很大程度上是由于它出色支持了分布式开发。

Mercurial http://www.selenic.com/mercurial/

这是一个跨平台的分布式系统，与 Git 系统有相似的目标，特别是它对分支的良好支持。

Bazaar http://bazaar-vcs.org/

它被设计为恰好够用，适应你的团队的工作流而不是强加自己的模型。

Bugzilla http://www.bugzilla.org/

在很长一段时间里，作为 Mozilla 项目一部分而开发的 Bugzilla 是问题追踪系统的默认开源方案。然而，最近人们有了更多的选择。

Trac http://trac.edgewall.org/

Trac采取极简主义的方法，旨在尽量避开开发人员。它同与自己的wiki网站的结合效果是尤其著名的。

Redmine http://www.redmine.org/

相对较晚进入人们视野的 Redmine 似乎能够被很好地支持，并且目前进展良好。

开源解决方案的传统弱项是在开发环境中对源代码控制和问题跟踪两者的结合。最近，这样的状况在 IDE 中有了相当程度的改进，例如，Eclipse 为 Subversion 提供了出色的支持。

A.1.2 托管解决方案

SourceForge http://sourceforge.net/

SourceForge是为开源项目提供托管的众多网站中最著名的一个。它们集合了大量工具，例如源代码控制、问题跟踪、文档记录工具等等。其他还包括Google Code（http://code.google.com/hosting/）和特定语言的站点，例如RubyForge (http：//rubyforge.org/)。

GitHub http://github.com/

GitHub 提供 Git 的托管，最近由于开始托管 Ruby on Rails 项目而备受瞩目。

Lighthouse http://lighthouseapp.com/

Lighthouse 是一个综合了 Subversion 和 Git 的可托管的问题追踪系统。

Unfuddle http://unfuddle.com/

这是一个安全的托管项目管理解决方案，提供 Subversion 和 Git 的托管，并集成了问题追踪系统。

Rally http://www.rallydev.com/

Rally 提供了敏捷的生命周期管理工具。

VersionOne http://www.versionone.com/

这是一个为敏捷软件开发而特别设计的项目管理和规划工具。它在托管的同时也提供本地安装。

Pivotal Tracker http://www.pivotaltracker.com/

Tracker 是一个免费、屡获殊荣的、基于故事的项目规划工具，能够使团队进行实时协作。

A.1.3 商业解决方案

Perforce http://www.perforce.com/

Perforce 是一个尤其关注跨平台支持和性能的源代码控制系统。它还包括一个简单的问题追踪系统，也可以结合多种开源或商业解决方案。

FogBugz http://www.fogcreek.com/FogBugz/

Fog Creek 公司的 FogBugz 是一个灵活的缺陷追踪和项目规划工具，可以本地安装或作为托管解决方案。传统上用于 Windows 系统，但正在被移植到 Linux 和 Macintosh 平台。

Visual Studio Team System http://msdn.microsoft.com/teamsystem/

微软的 Visual SourceSafe 长期以来都是众矢之的，由于很多失败的案例而饱受批评。公平地说，微软公司对此应无话可说。尽管公司有使用自己产品的政策，但是看起来它们似乎没用过这个软件。幸运的是，微软公司一直没有停止改进这款提供了完全集成的源代码控制及项目管理解决方案的系统。

Rational ClearCase 及 ClearQuest http://ibm.com/software/awdtools/clearcase/

ClearCase 源代码控制系统，以及与它结合的问题追踪解决方案 ClearQuest，过去常被认为是

企业的默认选择。然而，它们昂贵且复杂，它们只适合拥有专门支持机构的大型团队。

StarTeam http://www.borland.com/starteam/

这是一个完全集成的源代码控制和项目管理系统。

BitKeeper http://www.bitkeeper.com/

这是一个和 Git 有相似目标的分布式系统。

A.2 构建和持续集成工具

我们已经详细地考察了构建过程自动化的好处，正如你所期望的那样，有许多现成的工具可以帮你完成这些工作。

A.2.1 构建工具

构建工具的鼻祖是久负盛名的 make。事易时移，现在有了一些更好的工具可供选择。

GNU Make http://www.gnu.org/software/make/

尽管基于 make，GNU Make 仍支持数量可观的拓展，允许对构建过程进行更精密的控制。

Autoconf http://www.gnu.org/software/autoconf/

Autoconf 尤其适合需要支持在大范围的不同环境中进行构建的开源软件。它允许构建系统自动决定主机系统上的哪个设备可用并进行相应的工作。

Jam http://www.perforce.com/jam/jam.html

Jam 通常只要求较少配置就能完成指定项目。

Boost.Build http://www.boost.org/doc/tools/build/

建立在 Jam 之上的 Boost.Build 提供一个标准的构建系统，尤其适合托管 C++软件。

SCons http://www.scons.org/

这是一个综合了与 autoconf 相似功能的 make 的替代品。

Ant http://ant.apache.org/

这是一个 make 的替代品，它已在 Java 世界中成为事实上的标准构建工具。

Maven http://maven.apache.org/

Maven 是一个不仅仅简单地管理构建过程的软件项目管理工具，它为 Java 世界带来包的管理、部署等功能，并且正在快速从 Ant 获取更多用户的认同。

Capistrano http://www.capify.org/

就其本身而言并不是构建工具，Capistrano 在大量不同的服务器上处理部署的任务。尽管与 Ruby on Rails 结合得特别紧密，它仍可以用于部署使用任何技术制造的产品。

A.2.2 持续集成工具

许多我们已经讨论的专有系统（比如微软的 Visual Studio Team 系统）都带有它们自己的持续集成解决方案。此外，还有一些可用的开源系统。

CruiseControl http://cruisecontrol.sourceforge.net/

这大概是最著名的开源持续集成系统。除了主要的 Java 实现，还存在.NET 和 Ruby on Rails 的版本。

Hudson http://hudson.dev.java.net/

这是一个开放源码的 J2EE 持续集成服务器。

A.3 有用的库文件

并非所有的工具都是独立的，很多工具，包括本节中提到的，都以库文件的形式存在，我们需要将它们与我们自己的代码进行连接。

A.3.1 测试

过去几年，测试框架雨后春笋般地大量涌现，其中许多都参照了 JUnit。这里我不可能把所有的工具都讲到，所以我只提 2 个在 Java 社区中最具分量的工具。

JUnit http://www.junit.org/

它是库文件的始祖。

TestNG http://testng.org/

这是最新的一个测试框架，它是建立在 JUnit 基础之上的，但是采用了一些不同的方法，并

获得了广泛的使用。

A.3.2 调试内存分配器

正如我们在 10.2 节中讨论的，在像 C 和 C++这样不提供内存管理的语言中，调试内存分配器是一个避免内存泄漏、损坏和其他常见的问题的不可或缺的工具。

libcwd http://libcwd.sourceforge.net/

这是一个开放的源代码调试支持库，它提供内存调试等一系列功能。

Microsoft Visual C++ http://msdn.microsoft.com/visualc/

微软的 Visual C++附带了一个内置调试内存分配器，欲知详情，请在 VC 文档中搜索“内存泄露检测和隔离”（memory leak detection and isolation）。

Mudflap http://gcc.gnu.org/wiki/Mudflap_Pointer_Debugging

Mudflap 是一项被构建到某些版本的 GNU C 和 C++编译器中的技术，它可以检测所有危险的指针和数组取值操作、一些标准库的字符串函数和堆函数，以及其他一些相关结构的范围和有效性测试。

Dinkumware http://www.dinkumware.com/

Dinkumware 销售包含广泛支持内存调试的 C 和 C++标准库。

Electric Fence http://perens.com/works/software/ElectricFence/

它使用虚拟内存硬盘来检测内存覆盖与再利用释放的内存。

A.3.3 日志

日志框架可让代码提供可配置日志的能力，能够在运行时根据单个特征对它进行启用、禁用、增加细节。

log4j http://logging.apache.org/log4j/

Apache log4j 大概是最知名的 Java 日志库，而且可以移植到现在的大多数主要语言中。

Logback http://logback.qos.ch/

Logback 由 log4j 的创始人 Ceki Gülcü 设计，作为 log4j 的继承者。

java.util.logging http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/

自 1.4.2 起，Java 包括一个标准的日志 API，java.util.logging，一般称为 JUL。

SLF4J http://www.slf4j.org/

The Simple Logging Facade for Java 通过提供一个可以在部署的时候给予不同实现的通用接口，来试图统一过多的 Java 日志 API。

syslog-ng http://www.balabit.com/network-security/syslog-ng/

syslog-ng 是最受欢迎的 BSD syslog 协议的实现方案，允许将数据从许多不同系统中整合到一个中心资料库系统以及丰富的基于内容的过滤器。

A.4 其他工具

最后，这是每一个开发人员可以使用的其他工具的快速一览。

A.4.1 测试工具

FitNesse http://fitnesse.org/

FitNesse 是一个验收测试工具，它允许将测试表现为表格形式的输入数据和预期的输出数据，在 *Fit for Developing Software: Framework for Integrated Tests* 一书[MC05]中有详细描述。

Watir http://wtr.rubyforge.org/

Watir 是一个自动化 Web 浏览器的开源库，它可以对 Web 应用程序进行自动化测试。它最初是用在 Windows 的 IE 浏览器中的，但现在正在开发其他浏览器的版本。

Selenium http://selenium.openqa.org/

Selenium 是一个用来自动化构建 Web 应用程序测试的跨平台的工具套件。

Sahi http://sahi.co.in/

Sahi 是一种针对运行在代理服务器上的 Web 应用程序的自动化测试工具。

The Grinder http://grinder.sourceforge.net/

这是一个使用 Jython 作为开发脚本的开源负载测试工具。

JMeter http://jakarta.apache.org/jmeter/

这是一个使用 Java 作为开发脚本的开源负载测试工具。

QuickTest Professional 和 LoadRunner http://www.hp.com/

QuickTest Professional 是一个自动化的功能 GUI 测试工具，LoadRunner 是一种性能和负载测试产品。

Peach Fuzzing Platform http://peachfuzzer.com/

Peach 是一个兼具生产功能和变异功能的模糊器。

RFuzz http://rfuzz.rubyforge.org/

RFuzz 是一个 Ruby 库，可以很容易地对 Web 应用程序进行模糊测试。

A.4.2 运行时分析工具

Valgrind http://valgrind.org/

Valgrind 是一个针对 Linux 的工具框架，包含内存分析和性能分析工具等许多东西。

BoundsChecker http://www.compuware.com/products/devpartner/visualc.htm

BoundsChecker 是 Compuware's DevPartner for Visual C++ BoundsChecker Suite 的一部分。它分析正在运行的程序来检测内存等问题。

Purify http://www.ibm.com/software/awdtools/purify/

IBM 的 Rational Purify 能在运行的程序内检测内存泄露和内存损坏问题。

DTrace http://opensolaris.org/os/community/dtrace/

DTrace 是一个备受推崇的动态追踪框架，它是由 Sun 开发的用于解决内核和应用程序问题的工具。它也被纳入了 Mac OS X“美洲豹”系统，包括 Instruments GUI。

A.4.3 网络分析器

如果你的软件依赖于网络通信（很难找到不依赖于网络通信的软件了），看看到底通过网络传输了什么是非常有用的。

一个网络分析器（有时也被称为数据包嗅探器）位于网络之中，用来捕获和分析所有流经

网络的数据包。然后，你可以过滤这些数据包而且只提取那些你感兴趣的数据，检查其内容。大致来说，一个包嗅探器是一种底层工具。它可以捕获所有的网络流量，但不一定要深入了解使用的协议。所以，如果通讯内容被加密了，数据包嗅探器是不可能有能力来显示被交换的信息的。

TCPDUMP http://www.tcpdump.org/

TCPDUMP是一个广泛使用的开源数据包嗅探器。

Wireshark http://www.wireshark.org/

Wireshark（以前称为Ethereal）是一个开源工具，它提供了类似TCPDUMP的功能，但它有一个图形化的前端界面与可供广泛使用的内置分析工具。

A.4.4 调试代理

调试代理是一个比网络分析器更高级的工具，它可以针对特定的协议进行分析。通常需要对你的软件进行不同的配置，以便它可以过代理而不是直接通信，经常这样做可以对通信内容进行更深入的分析。一些调试代理甚至可以查看加密的数据。

Charles http://www.charlesproxy.com/

Charles是一个跨平台的HTTP代理，它还支持对加密的通信内容进行调试。

Fiddler http://www.fiddlertool.com/

Fiddler是一个Windows HTTP代理，正如其名称所示，可以让你"窃取"传入或传出的数据。

A.4.5 调试器

在大多数情况下，你选择的调试器由你选择的编程语言、IDE或工具链所管理，所以要在这里提供一个选择清单是没有什么价值的。但是，有一个特殊的调试器我必须说一说。

Firebug http://getfirebug.com/

通过提供显著改进的客户端调试工具，Firebug 改变了 Web 开发。它可以让你检查和编辑DOM和CSS，并且可以监测和分析网络活动的性能，并提供对JavaScript调试的完整支持。

附录 *B*

参考书目

[Bec02] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley, Reading, MA, 2002.

[Bro95] Frederick P. Brooks, Jr. *The Mythical Man Month: Essays on Software Engineering*. Addison-Wesley, Reading, MA,anniversary edition, 1995.

[Car71] Lewis Carroll. *Through the Looking-Glass, and What Alice Found There*. Macmillan, 1871.

[Coc06] Alistair Cockburn. *Agile Software Development: The Cooperative Game*. Addison Wesley Longman, Reading, MA, second edition, 2006.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley Longman, Reading, MA, 1999.

[Fow] Martin Fowler. Mocks aren't stubs. http://www.martinfowler.com/articles/mocksArentStubs.html.

[HT00] Andrew Hunt and David Thomas. *The Pragmatic Programmer:From Journeyman to Master*. Addison-Wesley, Reading,MA, 2000.

[iet99] Hypertext transfer protocol – http/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.txt, 1999.

[Knu74] Donald E. Knuth. Structured programming with go to statements. *ACM Comput. Surv.*, 6(4):261–301, 1974.

[Lad03] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., 2003.

[MC05] Rick Mugridge and Ward Cunningham. *Fit for Developing Software: Framework for Integrated Tests*. Prentice Hall PTR,Englewood Cliffs, NJ, 2005.

[OW07] Andy Oram and Greg Wilson, editors. *Beautiful Code: Leading Programmers Explain How They Think*. Theory in Practice.O'Reilly Media, Inc., Sebastopol, CA, 2007.

[ray] The jargon file. http://catb.org/jargon/.

[Ray01] Eric S. Raymond. *The Cathedral and The Bazaar*. O'Reilly& Associates, Inc, Sebastopol, CA, 2001.

[Sch04] Ken Schwaber. *Agile Project Management with Scrum*.Microsoft Press, Redmond, WA, 2004.

[Sut99] Herb Sutter. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley, Reading,MA, 1999.

[Swi08] Travis Swicegood. *Pragmatic Version Control using Git*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX,2008.

[wCA04] Kent Beck with Cynthia Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA,second edition, 2004.

[Zac94] G. Pascal Zachary. *Show Stopper!: The Breakneck Race to Create Windows NT and the Next Generation at Microsoft*.Little, Brown, 1994.

